(2)

ETL-0528

AD-A209 132

Parallel Vision Algorithms

Second Annual Technical Report

Hussein A. H. Ibrahim, Editor
John R. Kender
Lisa G. Brown

Department of Computer Science
Columbia University
New York, New York 10027

January 1989

Prepared for:

Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209-2308

U.S. Army Corps of Engineers
Engineer Topographic Laboratories
Fort Belvoir, Virginia 22060-5546

Destroy this report when no longer needed.
Do not return it to the originator.

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

The citation in this report of trade names of commercially available products does not constitute official endorsement or approval of the use of such products.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release; Distribution is Unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CUCS-415-89	5. MONITORING ORGANIZATION REPORT NUMBER(S) ETL-0528
6a. NAME OF PERFORMING ORGANIZATION Columbia University 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION U.S. Army Engineer Topographic Laboratories
6c. ADDRESS (City, State, and ZIP Code) Computer Science Department New York, NY 10027	7b. ADDRESS (City, State, and ZIP Code) Fort Belvoir, Virginia 22060-5546
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Defense Advanced Research Projects Agency 8b. OFFICE SYMBOL (If applicable) ISTO	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DACA76-86-C-0024
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Boulevard Arlington, VA 2209-2308	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Parallel Vision Algorithms Second Annual Technical Report

12. PERSONAL AUTHOR(S)
Hussein A. H. Ibrahim, Editor; John R. Kender and Lisa G. Brown

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Annual	FROM 10/1/87 TO 12/28/88	1989, January	77

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Computer Vision, Artificial Intelligence, Image Understanding, Multi-Resolution, Stereo, Texture, Strategy Computing

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The "Parallel Vision Algorithms" second annual technical report covers the project activities during the period from October 1, 1987, through December 28, 1988. The objective of this project is to develop and implement, on highly parallel computers, vision algorithms that combine stereo, texture, and multi-resolution techniques for determining local surface orientation and depth. Such algorithms will immediately serve as front-ends for autonomous land vehicle navigation systems. During the second year of the project, efforts have concentrated on the following: first, implementing and testing on the Connection Machine the parallel programming environment that will be used to develop, implement and test our parallel vision algorithms. Second, implementing and testing multi-resolution stereo, and texture algorithms in this environment. Also, we continue our efforts for the refinement of techniques used in our texture algorithms. This report describes the status and progress of these efforts. We describe first the programming environment implementation, and how to use it. Then, we present algorithms and test results for multi-resolution stereo, and texture algorithms. More results of the efforts of integrating stereo and texture algorithms are presented.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL George Lukes	22b. TELEPHONE (Include Area Code) (202) 355-2732 / 22c. OFFICE SYMBOL CEETL-RI

DD Form 1473, JUN 86 Previous editions are obsolete. SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

Parallel Vision Algorithms

Second Annual Technical Report

Contract DACA76-86-C-0024

John R. Kender, Principal Investigator
Hussein A. H. Ibrahim, Research Scientist
Lisa G. Brown, Research Programmer
Department of Computer Science
Columbia University
New York, N.Y. 10027
(212)854-2736

Technical Report - CUCS - 415 - 89

Summary

The "Parallel Vision Algorithms" Second Annual Technical Report covers the project activities during the period from October 1, 1987 through December 31, 1988. The objective of this project is to develop and implement, on highly parallel computers, vision algorithms that combine stereo, texture, and multi-resolution techniques for determining local surface orientation and depth. Such algorithms can serve as front-end components of autonomous land vehicle vision systems. The activities of the first year of this account are reported in the first annual technical report "Parallel Vision Algorithms - Annual Technical Report" [9]. During the second year of the project, efforts have concentrated on the following: first, implementing and testing on the Connection Machine the parallel programming environment that will be used to develop, implement and test our parallel vision algorithms; second, implementing and testing primitives for the multi-resolution stereo and texture algorithms in this environment. Also, we continued our efforts to refine techniques used in our texture algorithms, and to develop a system that integrates information from several shape-from-texture methods. This report describes the status and progress of these efforts. We describe first the programming environment implementation, and how to use it. We summarize the results for multi-resolution based depth interpolation algorithms on parallel architectures. Then, we present algorithms and test results for the texture algorithms. Finally the results of the efforts of integrating information from various shape-from-texture algorithms are presented.

Preface

This report, submitted to the Defense Advanced Research Projects Agency (DARPA) of the Department of Defense (DOD), in response to Contract DACA76-86-C-0024, presents the progress during the second year of the "Parallel Vision Algorithms" project at Columbia University. The "Parallel Vision Algorithms" project is sponsored by DARPA as part of its Strategic Computing Program and administered by the U.S. Army Engineer Topographic Laboratories (ETL).

The objective of this project is to develop and implement, on highly parallel computers, vision algorithms that combine stereo, texture, and multi-resolution techniques for determining local surface orientation and depth. Such algorithms can serve as front-end components of autonomous land vehicle vision systems.

This report is prepared for the U.S. Army Engineer Topographic Laboratories, Fort Belvoir, Virginia, and the Defense Advanced Research Projects Agency, 1400 Wilson Boulevard, Arlington, Virginia under contract DACA76-86-C-0024. The ETL Contracting Officer is Mary Lu Williams. The ETL Contracting Officer's representative is George Lukes. The Program manager is LTC Robert Simpson. Questions regarding this document should be forwarded to Prof. John Kender, (212-854-8197.)

Table of Contents

List of Figures

1. Introduction

The objective of this project is to develop and implement, on highly parallel computers, integrated parallel vision algorithms that combine stereo, texture, and multi-resolution techniques for determining local surface orientation and depth. Such algorithms are envisioned as potential front-end components of autonomous land vehicle vision systems. During the first year of the project, efforts concentrated on two fronts; first, developing and testing the parallel programming environment used to develop, implement, and test our parallel vision algorithms; second, developing and testing multi-resolution stereo, and texture algorithms. In the second year of the projects, efforts have been concentrated on the implementation of the developed algorithms and techniques on the Connection Machine. Also, research has continued on refining our developed texture algorithms, on implementing integrated systems of stereo and texture on sequential machines, and on improving the ability to recognize surfaces in an image using fusion of information from more than one surface recognition module. This report describes the progress of these efforts in the second year. We describe first the implementation of the programming environment. Then, algorithms and test results for multi-resolution stereo, and texture algorithms are presented. Progress in integrating the results of different texture modules is then described.

The initial plans called for the testing and implementation of the parallel algorithms on the NON-VON Supercomputer (which was being developed at that time). With the NON-VON project being terminated, the Connection Machine was chosen as the target machine to develop and test our algorithms. An account was obtained on CM1 and CM2 at Syracuse University, and we used these machines to implement some of the developed algorithms.

2. Pyramid Programming Environment for Multi-Resolution Algorithms

Hussein Ibrahim and Lisa Brown

In this chapter, we describe the pyramid environment that has been implemented on the Connection Machine to program multi-resolution algorithms. The mapping scheme and the efficient simulation of a pyramid architecture on the Connection Machine have been explained in the first annual report [9], and in [10]. An implementation of this environment on the Connection Machine at Syracuse Northeast Parallel Architectures Center (NPAC) has been carried out during the second year of the project. This implementation included all the functions required for the mapping scheme and an implementation of the pyramid communication primitives and also implementation of pyramid loading and displaying functions. An environment that combine all these functions has been implemented and has been used to implement some of the multi-resolution algorithms for stereo and texture. In the following sections a brief description of the environment and the functions used is given. A user's manual has been written and is attached to this document [3].

2.1 Pyramid Emulation on The Connection Machine

This section describes a set of functions for using image pyramids on the Connection Machine. These functions are an extension of *LISP which itself is an extension of COMMON LISP. The functions were designed to work on the Connection Machine 2 of the Northeast Parallel Architectures Center located at Syracuse University.

2.1.1 Description of Pyramids

The routines described herein can be used to write *LISP programs for vision algorithms which use multi-resolution image pyramids to structure and manipulate image data [10]. Basically, by using these routines, the user can program multi-resolution algorithms on the Connection Machine so that inter-pyramid communications will be executed efficiently. Examples of typical multi-resolution algorithms include the computation of depth from stereo or motion and image registration. Students here at Columbia University and also at Syracuse University have used the environment to implement hierarchical stereo correlation. It is important to note that this system deals with pyramids where each node communicates with exactly four children below it, to a single parent above it, and to four neighbors on the same level. This system would probably be inappropriate for emulating pyramids with more general configurations.

2.1.2 Naming Conventions

The functions described all use a new data structure called a pyramid (or pmd). This structure is actually composed of two pvars (parallel variables in *Lisp language) although this is transparent to the user. All the functions are written so that in most cases those dealing with pyramids are analogous to standard *lisp functions which deal with pvars. For example, to create a pmd the function is *defpmd (like *defvar) and *set-pmd (like *set). In addition, the following conventions are adhered to throughout so that properties of functions and variable names are as obvious as possible:

- [pmd!!]: All functions which end with pmd!! return a pmd.
- [*--pmd]: All functions which begin with an asterisk and end with "pmd" have arguments which are pmds.
- [*pmd--*] All variables which start with "*pmd" and end with an asterisk are global pyramid variables (see section 2.1.4 on global variables).

These conventions are similar to those for pvars in *LISP. We also adhere to their conventions; namely, functions ending with !! return pvars, functions starting with asterisks use pvars internally.

2.1.3 Getting Started

To use the pyramid environment it is necessary that the following two conditions are met:

1. The number of logical processors should equal the number of physical processors. (This will be extended so that the number of logical processors can be any multiple of four times the number of physical processors.)
2. The machine should be configured for 2 dimensions Hence, the number of processors should be of the form 2^2n where n is an integer.

The first condition can be met by attaching to the same number of processors as you configure the machine when *cold-booting. The second condition is met with the :initial-dimensions to *cold-boot. To load the pyramid system, the file "pyramid-emulate" should be loaded and the function pyramid-emulate is then executed. An example session which shows a start-up is shown in the file "pmd-example." Most of the commands issued in this session are also contained in the file, "pmd-init.lisp" which is the initialization file used for testing the system.

2.1.4 Global Variables

The following are global variables used by the pyramid system which might also be useful to the user. A variable of particular importance is *pmd-level-number*.

- [*pmd-number-of-levels*]the number of levels in the pyramid.

- [*pmd-size*] the size of one side of the base of the pyramid
- [*pmd-self-address*] the address of each processor in the pyramid. Note: these addresses indicate the location of the physical hypercube connections that connect the processors.
- [*pmd-level-number*] a pvar which indicates the particular level other than the lowest level which a processor represents. This pvar can be used in a *when to select only the processors on a certain level (above the leaf level).

2.1.5 Pyramid Primitives

The following functions are for creating, allocating and setting the values of pyramids and their levels. They form the basis of all pyramid programs. In addition, the *LISP function *let can be used to dynamically create pmds using the function allocate-pmd!!.

```
(*defpmd pmdname &optional pmd-initialization)

(allocate-pmd!!)

(*deallocate-pmd pmd)

(*set-pmd pmd-1 pmd-2)

(*set-level-pmd level pmd-1 pmd-2)
```

2.2 Communications

In this section, we describe functions to emulate pyramid communication primitives. These include horizontal communications, communications between PE's on the same pyramid level; and vertical communications between PE's on successive levels of the pyramid.

2.2.1 Horizontal Communications

The following functions execute mesh communications within individual levels of the pyramid. They work according to the scheme specified in [10].

```
(shift-level-pmd!! level direction source-pmd &optional dest-pmd
                   &key border-pmd)
```

This uses the mesh on the specified level to shift the data in source-pmd in the specified direction ('e 'w 'n or 's) and puts the resulting level in the optionally specified dest-pmd and returns it. For example: (shift-level-pmd!! 1 'e pmd-in pmd-out :border-pmd zero-pmd) shifts level 1 in pmd-in to the east and stores the result in level 1 of pmd-out which is returned. The border-pmd is used in the same way the border-pvars are used in *lisp commands such as pref-grid!!.

```
(shift-pmd!! direction source-pmd &optional
                        dest-pmd &key border-pmd)
```

This is the same as the above function except all levels are shifted and stored in dest-pmd and returned.

2.2.2 Vertical Communications

The following functions execute top-down communications between levels of the pyramid. They work according the scheme specified in section 4 of the article "On Implementation of Pyramid Algorithms on the Connection Machine" [10]. Top-down communications either transfer (and combine) data from one or more children up the pyramid or from a single parent to one or more of its children. Children are specified as 'a 'b 'c or 'd. When communicating up the pyramid an operation is specified which indicates how the four children are combined before setting the parent value. The operation can be any parallel operation such as +!! or *!!.

```
(send-level-parent-pmd!! level operation source-pmd
                                              &optional dest-pmd)

(send-level-children-pmd!! level source-pmd &optional dest-pmd)

(send-level-child-pmd!! level child source &optional dest-pmd)

(ave-pmd!! level source-pmd &optional dest-pmd)
```

The command ave-pmd!! is used to make a pmd using an image stored in the lowest level and making each successive level above by averaging the four children of each parent. This is a typical example of a function which uses the vertical communication functions to make an image pyramid.

2.3 Pyramid Input/Output

These routines can be used to store a pyramid or pvar into a file for use with standard sequential image processing routines or for reading such a file into a pyramid or pvar for use with this system. Typically a pvar represents an image from which a pyramid can be constructed.

```
(read-pvar!! ''filename'' &optional pvar)

(read-pmd!! ''filename'' &optional pmd)

(write-pvar!! pvar ''filename'')

(write-pmd!! pmd ''filename'')
```

2.3.1 Pyramid Display

These routines can be used to print the data stored in a pyramid either as a single level or the entire pyramid.

```
(*display-level-pmd pmd level)

(*display-pmd pmd)
```

The *Lisp code implementation of this environment is included in the Appendix at the end of this report.

3. Depth Interpolation Problem - A Multi-Resolution Approach

Dong Jae Choi, John R. Kender

Research in depth interpolation on fine-grained Single Instruction Stream Multiple Data Stream (SIMD) machines such as NON-VON or the Connection Machine has been completed and has resulted in a Ph.D. thesis attached to this report [4]. The principal results of the work are a detailed comparison of five approaches to determining surface depth from sparse data, such as results from stereo. Two of the approaches are new, and are provably optimal. Extensive simulations written in a way that allows straight forward transfer to the Connection Machine has concluded that communication costs begin to dominate compute costs when algorithms are optimal. Additionally, it appears that pyramid (multi-resolution) approaches to the problem provide speedups of 2 to 200 times over conventional approaches. These results have been submitted to a journal, and have already appeared in several conferences.

In more detail the work covered the following: many constraint propagation problems in early vision, including depth interpolation, can be cast as solving a large system of linear equations where the resulting matrix is symmetric and positive definite (SPD). Usually, the resulting SPD matrix is sparse. The depth interpolation problem on a fine-grained single instruction multiple data (SIMD) machine with local and global communication networks has been solved. It has been shown how the Chebyshev acceleration and the conjugate gradient methods can be run on this parallel architecture for sparse SPD matrices. Using an abstract SIMD model, for several synthetic and real images it has been shown that the adaptive Chebyshev acceleration method executes faster that the conjugate gradient method, when given near optimal initial estimates of the smallest and largest eigenvalues of the iteration matrix.

These iterative methods have been extended through a multigrid approach, with a fixed multilevel coordination strategy. It has been shown again that the adaptive Chebyshev acceleration method executes faster than the conjugate gradient method, when accelerated further with the multigrid approach. Furthermore, it has been shown that the optimal Chebyshev acceleration method performs best since this method requires local computations only, whereas the adaptive Chebyshev acceleration and the conjugate gradient methods require both local and global computations.

4. Parallel Texture Algorithms

We believe, as Bajscy and Lieberman contend in their pioneering work in shape from texture [1], that texture is the most significant feature of outdoor scenes. Shape-from methods based on stereo or motion often have inherent difficulty dealing with highly textured scenes since feature matching becomes intractable. On the other hand, enormous information is available for explicit surface reconstruction where surfaces are textured; indeed, many surfaces cannot be unambiguously reconstructed unless they are textured. (See [2] for a good example.) Guidance and recognition tasks could be greatly improved with the assistance of a shape-from-texture system if they were able to deal with a broad range of natural textures without complex structural knowledge or extreme computational costs.

In this section of the report, we will describe two approaches that we have taken to exploit textural cues for the recovery of three dimensional information. The first approach, investigated by Lisa Brown, is based on our previously reported work on the projective foreshortening of isotropic texture autocorrelation. During the past year, this work has been extended to anisotropic textures using multiple viewing and an investigation has begun on exploiting integrability in reconstructing a depth map from surface orientation measurements. The second approach, taken by M. Moerdler and J. R. Kender, entails integrating several shape from texture algorithms. This work was based on a system designed during the first year of the project which fuses several conflicting and corroborating texture cues to derive surface orientations. During the past year, this system has been implemented, tested and simplified so that it is now a general method for incorporating new texture modules as they become available.

4.1 Surface Orientation for a Wide Class of Natural Textures Using Texture Autocorrelation

Lisa Brown

We report on a refinement of our technique for determining the orientation of a textured surface from the two-point autocorrelation function of its image [9]. In our initial approach we needed to assume textural isotropy which we now replace with knowledge of the autocorrelation moment matrix of the texture when viewed head on. The orientation of a textured surface can then be deduced from the effects of foreshortening on these autocorrelation moments. We have applied this technique to natural images of planar textured surfaces and obtained significantly improved results on anisotropic textures which under the assumption of isotropy mimic the

effects of projective foreshortening. The resulting method is capable of measuring surface orientation for a broader class of naturally occurring textures than has been previously been possible. We will describe briefly this new technique, its implementation and results and its potential practicality for autonomous navigation.

In our previous method and its precursor proposed by Witkin [20], local surface orientation is computed from the effects of foreshortening for textures which are assumed to be isotropic. By isotropic, we mean, statistically speaking, textures that have no inherent directionality. Unlike the majority of shape-from-texture methods, which rely on a texture gradient caused by the perspective projection, these methods look at how a statistical distribution, dependent on the direction of textural components in the image, is effected by the foreshortening due to orthographic projection. Witkin proposed to use a histogram of edge directions to determine surface orientation via a maximum likelihood fit, while in our previous work, we used the second order moments of the two-dimensional two-point autocorrelation. The latter has the advantage of being simpler and more robust, broadening the range of textured surfaces whose orientation could be determined because of the use of information from all parts of the image. Nevertheless, the assumption of isotropy is a very strong and limiting factor for both of these methods.

The method proposed here is an extension of our earlier technique based on the foreshortening of texture autocorrelation. In order to analyze how surface orientation could be obtained for a much broader class of textures, *a priori* information about each texture, specifically the autocorrelation moment matrix of the texture when viewed head on, is used. With this additional information, the original technique can be extended to all textures regardless of whether or not they are isotropic.

In the original method, it was possible to compute the surface orientation directly from the relation between the autocorrelation moment matrix when viewed head on $\mu_\perp$ and the current estimate, since $\mu_\perp$ was known to be a multiple of the identity since the texture was assumed to be isotropic. The slant and tilt could be specified as simple functions of the autocorrelation moment matrix. Without this assumption, further information is necessary to resolve the orientation. We have chosen to use the autocorrelation moment matrix for the texture when viewed head-on as an additional input. Given this prior information, it is now possible to compute the surface orientation by an iterative solution using Newton's method.

To test this method, a series of images were taken of three commonly found textures: brick, wood and stone. The brick and wood were both highly anisotropic while the stone was sufficiently anisotropic that the original method gave unsatisfactory results. In each case a single

planar surface was photographed from a sufficient distance that orthographic projection was a good approximation, and that the entire image consisted of an image with a single orientation. For each texture, several photographs were taken of each texture at varying surface orientations including one which was a head-on view. An attempt was made to keep the same location on the surface in the center of each photograph and to keep the camera at a fixed distance from the surface. The actual orientations were obtained, as in the previous study, using an identical picture for each orientation, in which a flat circular object was placed on the surface.

As before, photographs were digitized to yield 256 x 256 8-bit gray-scale images, and the autocorrelation was computed as the Fourier transform of the power spectrum of the image. Based on our previous results, to compensate for statistical noise, the second order moments were summed only over those autocorrelation values which were greater than the average value found in a ring of radius 10 pixels.

The nonlinear system of equations given was solved using Newton's Method. This was guided by an initial estimate of the foreshortening matrix determined from the solution obtained if we assume the surface is isotropic (i.e. we let $\mu_\perp$ be proportional to the identity.) Convergence with insignificant computational cost occurred in all instances except one in which the autocorrelation moment matrix was not positive definite, presumably due to statistical error. This was confirmed by another picture of the same texture (wood) whose orientation was similar. In this instance, the matrix was marginally positive definite, convergence was slower, but a good estimate of the orientation was found. Over the whole sample, the average error in the slant and tilt estimates was 8 and 4 percent respectively. The worst errors were 17 and 10 percent respectively. A 5 percent error is estimated in the measurement of the actual orientation.

Examples of two of the pictures and their autocorrelation are given in Figure 4-1. From the autocorrelations in this figure you can see that the textures are anisotropic. If the texture was isotropic the autocorrelation would be composed of concentric scaled elliptic iso-contours (which are circular if the surface is viewed head on.) For each view of a textured surface contained in the figure, one picture depicts the surface with a flat circular object laying upon it. This makes the surface orientation explicit just as it would for the autocorrelation if the texture were isotropic. Since the textures are anisotropic, the comparable orientation information is contained only in the autocorrelation relative to the head on autocorrelation. Our technique measures this distortion, due to foreshortening, which transforms the autocorrelation in the same way it transforms the image under orthographic projection.

In all but the previously mentioned case, surface orientation estimates were accurate even though

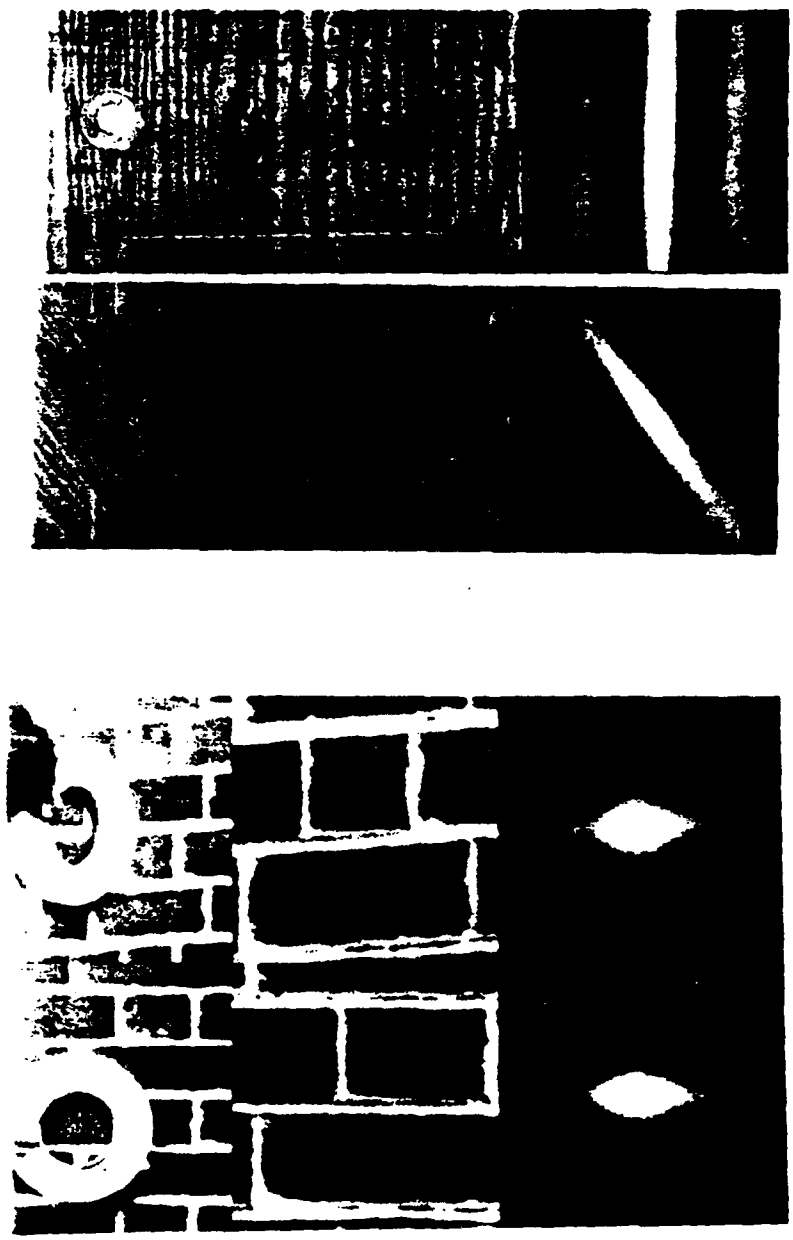


Figure 4-1: Example of Two Anisotropic Textures and Their Autocorrelation

in many cases textural anisotropy mimicked the effects of foreshortening. Use of prior knowledge of the autocorrelation moments from a head-on view was chosen for simplicity, but any other view would suffice as long as its orientation was known. Since it is not feasible to have prior information about each texture, in practice the next step would be to use multiple views of each texture in which the orientations of all the views are unknown. From the change in autocorrelation moment matrices, relative orientations could be computed which a surface reconstruction algorithm would ultimately fit into a coherent 3-D perception of the scene.

We have examined a refinement of our technique for determining the orientation of a textured surface from the two-point autocorrelation function of its image. The previous assumption of textural isotropy was replaced by knowledge of the autocorrelation moment matrix of the texture when viewed head on. The orientation was then deduced from the effects of the foreshortening on the autocorrelation moments. The new technique can successfully determine surface orientation for anisotropic textures. This technique suggests an image understanding system guided by a texture classification scheme would be capable of determining surface orientation for a broader class of textures than has been previously possible. The results of this work confirm that powerful cues for 3-D perception can be extracted from textured surfaces.

4.1.1 *Constructing* Depth Map From Surface Orientation Measures

In completing our work on shape from texture autocorrelation, we have been investigating techniques for reconstructing a depth map from surface orientation measurements. Several studies have been conducted on recovering depth from sparse depth and orientation data [8, 18, 5]. However, because our texture module is capable of computing surface orientations at every point in the image, our problem is no longer one of interpolation. Instead, we are interested in exploiting the integrability constraint using a simple Gauss-Seidel relaxation in order to recover a depth map of the original image. Our investigation began with an error analysis of the sensitivity of the depth map when the orientations have multiplicative Gaussian noise. Most recently, our research in this area has been an empirical study of enforcing integrability on shape from texture autocorrelation, using real imagery of a physically existing surface model whose depths are known accurately. Our objective has been to develop the first working shape-from-texture system which

- computes 21/2 D depths (not just orientations)
- works for real imagery (not synthetically projected textures)
- of natural textures (both naturally occurring and those found in nature)
- and has been tested against objective quantifiable measures.

By accomplishing this goal, this unit could then directly serve as part of a front-end for an

autonomous land vehicle navigation system.

Recently, related work by Frankot & Chellappa has shown the merits of enforcing the integrability constraint for constructing consistent depths for shape from shading. Their approach is to find the orthogonal projection onto the vector subspace spanning the set of integrable slopes represented by a set of basis functions using the POCS (projection of convex sets) method which they prove is appropriate for this domain. Shading information however, even when coupled with information about the light sources, is only capable of indicating slope unless other constraints such as smoothness are imposed. Since the problem of depth reconstruction in shape from shading is under constrained, integrability plays a different role than it does in shape from texture where including integrability makes the problem becomes over-determined. Yet, Frankot & Chellapa show that for a particular shape from shading algorithm enforcing integrability increases the speed of convergence, lessens the need for regularization and improves accuracy.

4.1.2 Experimental Results

In order to study the sensitivity of noise on orientations, when reconstructing the 2 1/2 D depth map, we looked at a variety of synthetic continuous differentiable surfaces. Our method of constructing these surfaces was to use the superposition of any number of '3D sine bumps' where a sine bump was simply a period of a sine wave from $-\Pi/2$ to $3\Pi/2$. This had the nice property that we could construct a wide range of surfaces but all the orientations are well-defined and easy to compute. In all cases, we chose a collection of sine bumps which composed a surface with zero boundaries although it would be interesting to test the sensitivity of the relaxation to variations in this constraint as well.

We then looked at how a straightforward Gauss-Seidel style relaxation would recover the depths from the orientations. Our objective function corresponded to the least squares minimization (or convolution kernal) shown in Figure 4-2(a). Notice the checkerboard-style locations of the orientations with respect to the locations of the depths. This is important in order to avoid a coupling which occurs in many other kernal choices. For example, consider the choice given in Figure 4-2(b) where the finite difference estimate of the orientations uses the four surrounding depths. In this case, the central depths (marked with circles in the figure) cannot be utilized by the resulting objective function because their effects are canceled out. Because in the checkerboard pattern, orientations and depths have different grid locations and we want the results at the same locations, the orientations for the reconstruction were computed by averaging the two neighboring orientations for each gradient direction. It was felt that the high frequency

information that would be lost in doing so would be extremely minimal in the case where texture autocorrelation was used since considerable overlap of the original image would have already been necessary to compute neighboring orientations.

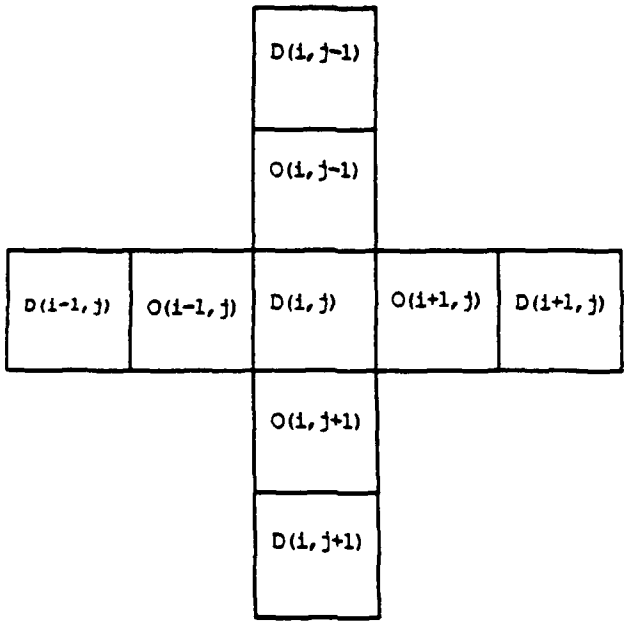
Examples of two reconstructed synthetic surfaces studied are given in Figure 4-3 and 4-4. In Figure 4-3 the surface has been accurately reconstructed from orientations with a significant amount of multiplicative Gaussian noise are shown. The method appears highly insensitive to noise, even when the standard deviation of the multiplicative Gaussian noise is as large as 1.6. We believe this is a result of the substantial smoothness inherent in the synthetic surface. In Figure 4-4, since the slopes are steeper, the resolution of the orientations is not high enough to recover the depth accurately. However, as can be seen in the figure, the reconstructed surface matches the true surface except for a high frequency component. A smoothness constraint added to the objective function would probably be sufficient to correct for this in cases where it would be possible to make this assumption.

It was our original objective to complete a working shape from from texture system applicable to real imagery of natural textures. We wanted to see what the potentials and shortcomings would be in the real world of lens blurring and distortions, texture anomolies, small surface perturbations, slight lighting variations and other uncontrollable and unknown factors. Lastly, we wanted to be able to quantify our results against an objective accurate measure. To this end, we decided to build a surface model, satisfying the set of assumptions which would allow us to apply our depth reconstruction program unmodified yet still use true texture and real imagery and have an objective ground truth. The assumptions we needed to address were texture isotropy, zero boundaries and smooth surfaces. We built a model, approximately two feet square and six inches high, composed of a 'blanket' of paper-mache, with wood cuttings glued on as texture. We found these materials enabled us to satisfy our requirements, since in addition to isotropy, zero boundaries and smoothness, we would be able to accurately measure the true depths using a 3-D sonic digitizer, and easily acquire a high resolution photograph from a sufficient distance to approximate orthographic projection (20 Meters). The enlarged photographed was then scanned at 300 dpi to give us a large data set with which to recover known depths.

4.2 Conclusions

In summary, we have shown that the method has performed well on synthetic data. We are still testing this algorithm on the real images acquired from our surface constructed as a test case. We will then compare these results with depths measured directly. If we can successively recover

depth in this fashion, we believe we will have completed the first shape from texture system to provide true shape (depth information) from real images of naturally textured surfaces verified against objective measurements.



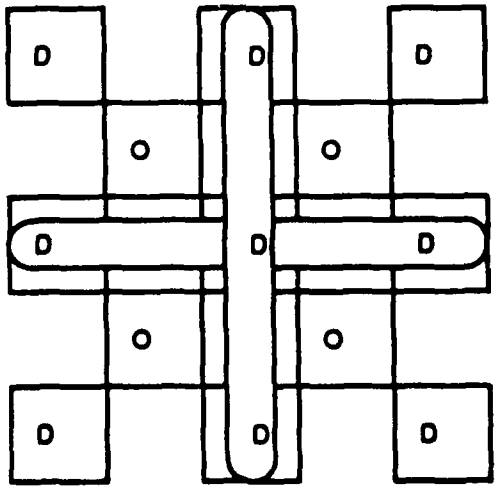
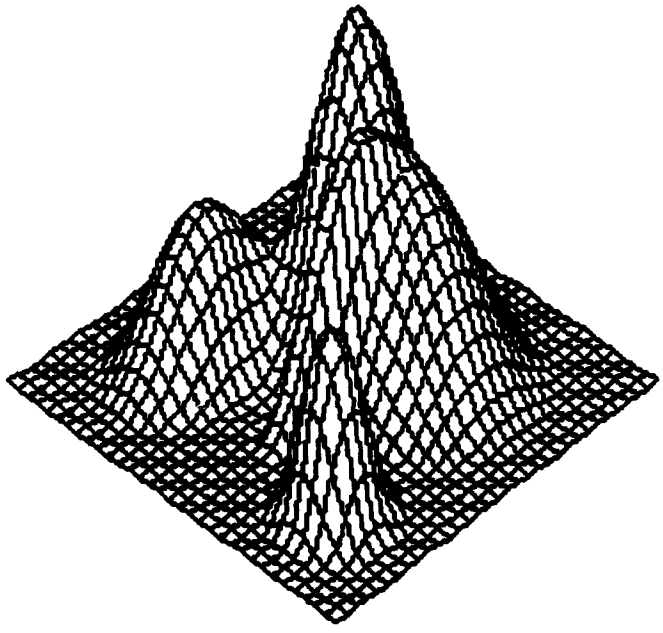


Figure 4-2: Relation of Depths and Orientations for Objective Function
(a) shows a good relationship for effectively using the depth information
(b) shows a case where the central depths (circled) become coupled and their effects canceled.



Standard Deviation	*Iterations*	*Ave. Error*
0	1701	.0537
.2	1701	.0545
.4	1700	.0577
.8	1691	.0743
1.6	1690	.1691

Figure 4-3: Example of Smooth Reconstructed Surface. Table shows the change in the average error in the reconstruction for different amounts of multiplicative Gaussian noise whose standard deviations are given. Also shown are the number of iterations required for the relaxation to reach a fixed rate of change.

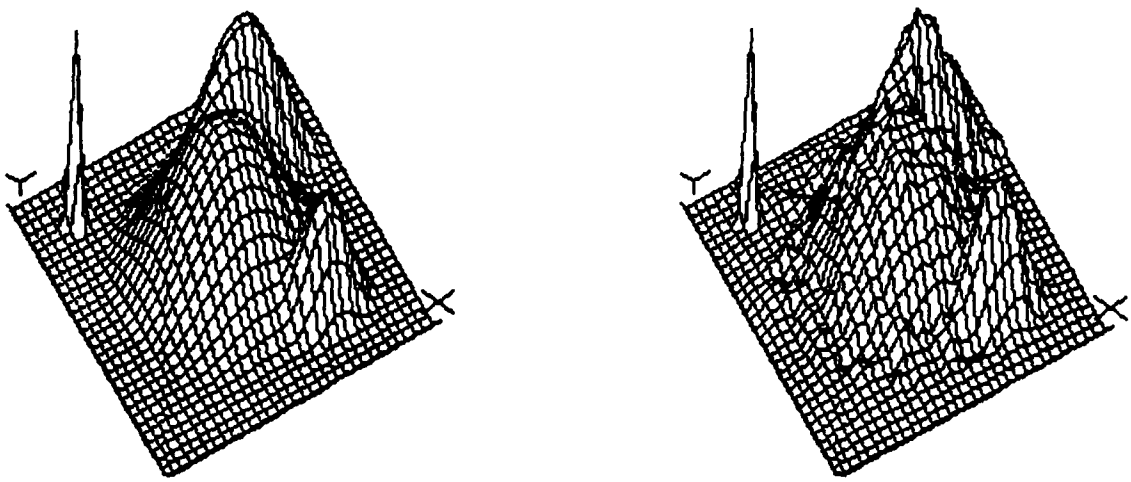


Figure 4-4: Example of Reconstructed Surface with Significant Noise added to Orientation Measurements. Surface on the left shows the true depths. Surface on the right is the reconstruction. Notice the high frequency error in the reconstruction.

4.3 An Integrated System That Unifies Multiple Shape From Texture Algorithms

Mark L. Moerdler and John R. Kender

This section describes an approach to the classification and segmentation of textured imagery. The approach utilizes information derived during the recovery of surface shape parameters, which are generated by a system that integrates multiple shape-from cues.

The robustness of this approach is illustrated by a system that integrates several shape-from-texture cues, recovering not only shape information but also segmenting images into surfaces and aiding in the classification of the surface textures. One example is given of the system operating on real, camera-acquired imagery.

The multiple shape-from-texture paradigm, first proposed by Moerdler [14, 15], can aid in texture analysis because of the way in which shape-from-texture cues function. Specifically, texture can be used to derive the shape of surfaces if a priori assumptions are made about the surface texture. Individually, shape-from-texture cues are limited by their underlying assumptions, yet if integrated into a large system comprised of multiple cues, they allow the surface orientation to be recovered. The specific cues that are used to derive the chosen surface orientation can be found and their underlying assumptions used to restrict the class of possible textures. If a large enough group of shape-from-texture methods is used, then the texture analysis problem becomes greatly constrained [15].

The surface segmentation problem can be simplified utilizing information generated during the operation of a multiple shape-from-texture system. In such a system, each orientation constraint is generated using more than one texture element (or texel). Once the surface parameters have been recovered the constraints can be re-analyzed and the texels that were used to generate the surface parameters can be combined into groupings. As will be shown below, these groups are a first approximation to surface segmentation. Additional segmentation information can be supplied by the texture analysis component, if the various surfaces in the image differ in their textures.

The robustness of this approach is illustrated by a system that integrates three shape-from-texture cues: shape-from-uniform-texel-spacing [11], shape-from-uniform-texel-size [16], and shape-from-virtual-parallel-lines [15]. These three cues generate orientation constraints for different overlapping classes of textures, thus limiting the texture analysis problem and aiding in surface segmentation. It is important to note that any of a large range of shape-from-texture cues could

be integrated in an operation system, thus increasing the robustness of the system's shape recovery, texture classification, and surface segmentation abilities.

4.3.1 Design Methodology

In this section, we will summarize the design methodology. Additional detail can be found in the annual report [9]. At the center of a texture analysis and image segmentation system is the shape recovery component. This component, based on the interaction between multiple shape-from-texture cues, effectively drives texture analysis and image segmentation. The cues that comprise the shape-from-texture component are based on assumptions about both the texture and the surface segmentation. These assumptions are tested during the constraint fusion phase and information about the type of texture and the surface segmentation are generated.

We first discuss how the integration of multiple shape-from-texture cues derives surface shape parameters. We then describe how the underlying assumptions of the shape-from-texture cues can be used to classify the surface texture while creating a first approximation to the combination of textured surface patches into surfaces.

The generation of orientation constraints from perspective distortion is performed using one or more image texels. The orientation constraints can be considered as local, defining the orientation of individual surface patches (called *texel patches*[*]) each of which covers a texel or group of texels. This definition allows a simple extension to the existing shape-from-texture methods beyond their current limitation of planar surfaces or simple non planer surfaces based on a single texturai cue. The problem can then be considered as that of intelligently fusing the orientation constraints per patch. This process can be broken down into three phases:

1. creation of texel patches and multiple orientation;
2. computation of constraints for each patch;
3. unification of orientation constraints per patch into a "most likely" orientation.

During the first phase, the different shape-from-texture components generate texel patches and *augmented texels*. Each augmented texel consists of the 2-D description of a texel patch and a list of weighted constraints on its orientation. The orientation constraints for each patch are

[*]Texel patches are defined by how each method utilizes the texels. Some methods (e.g. uniform texel size) use a measured change between two texels; in this case the texel patches are the texels themselves. Other methods (e.g. uniform texel density) use a change between two areas of the image; in this case the texel patches are these predefined areas.

potentially inconsistent or incorrect because the shape-from methods are applied to noisy images, are locally based, and derive constraints without a priori knowledge of the type of texture or number of surfaces.

In the second phase, all the orientation constraints for each augmented texel are consolidated into a single "most likely" orientation by a Hough-like transformation on a tesselated Gaussian sphere. During this phase the system will also merge together all augmented texels that cover the same area of the image. This is necessary because some of the shape-from components define "texel" similarly, thus the constraints generated should be merged and a single orientation generated for the surface patch. In those instances where more than one "most likely" orientation is found, the system re-analyzes the texel's constraints and checks their validity as measured in relationship to other texels. This allows it to prune out some of the constraints and possibly remove one or more of the "most likely" orientations resulting in a single orientation. This is a type of symbolic segmentation where segmenting is applied to the "intrinsic image" of surface orientation.

Once the individual oriented surface patches have been found, the system re-analyzes the orientation constraints to recover the *valid constraints*. The system uses the valid constraints in both simplifying texture analysis and surface segmentation. The term ''valid constraints'' denotes those constraints that were used to generate a solution rather than connoting any world knowledge that the constraints are correct.

The valid orientation constraints are determined individually for each surface patch. The constraints on the orientation of each surface patch, as stored in the augmented texel, are compared to the patch's orientation. Each constraint that fulfills the orientation, within an approximation, is considered as valid. This approximation is based on the quantization with which the orientation was originally computed*.

4.3.2 Recovering Texture Classification

Texture classification is performed by computing which of a group of features describe a given texture. As yet no single set of features differentiates all possible textures [13]. Instead, researchers have proposed different types of features for specific classes of textures. It should be noted that some of the features that have been proposed are ad hoc in nature rather than based on intrinsic properties of ''texture.''

---

*The Gaussian sphere is tesselated and any surface orientations generated using it are approximate.

The underlying assumptions of existing shape-from-texture cues limit the class of textures to which the cue is applicable. If it can be established which cue is applicable to a specific texture, it is in effect equivalent to deriving the texture classification. Since this classification information is unavailable prior to deriving the surface shape, the multiple shape-from-texture paradigm applies all of the cues to the image. After the knowledge fusion phase the system is able to determine the valid constraints and therefore which assumption are valid for each texel.

The underlying assumptions of shape-from-texture cues attempt to model the intrinsic properties of texture (e.g. uniform space, uniform size etc. [7]). The texture classification algorithm above is therefore based on a model of texture. If a large number of cues are used, the texture model becomes more complex and is better able to describe both natural and synthetic texture.

If surface segmentation is performed at the same time as texture classification then not only can the image be partitioned into regions but a texture model can be generated for each region in the image.

4.3.3 Approximating Surface Segmentation

Previous texture based segmentation algorithms [12, 17] partitioned images into regions based on differences in the image texture. The difference is defined as a measured change in some feature or features of the texture (also called texture measures). Depending on the specific group of features used, an image may be segmented into different regions.

Since these methods use only texture classification based information to partition the image, the segmented regions do not necessarily correspond to surfaces. If the features are either too sensitive or do not really model the world, the surfaces will be partitioned into multiple regions. At the other extreme, if the features do not correspond to all of the attributes of the texture then regions of the image may contain more than one surface. An additional handicap of these texture segmentation algorithms is that they are unable to correctly partition images containing overlapping transparent textured surfaces.

Perceived texture (camera or otherwise acquired) is the product of numerous physical processes. If segmentation is to be more exact then it should consist of more than measured changes in attributes of the texture. The multiple shape-from-texture paradigm, described above, can easily be extended to aid in surface segmentation. Implicit in the constraint generation component of the algorithm is the assumption that the image contains a single textured surface. This assumption manifests itself in the generation, by each of the cues, of constraints on the orientation of all possible groups of texels. This is necessary since no a priori segmentation

information is available.

So long as texture is assumed not to mimic perspective distortion effects, texel groupings that cross surface boundaries will create pseudo-random orientation constraints. If a number of correct constraints are generated, then these pseudo-random constraints will not affect the system in choosing the correct constraints and thus will not be included in the valid constraints.

Once the valid constraints have been recovered, after the system has selected the surface orientation parameters, a segmentation of the image can take place. Each valid constraint is generated by one of the shape-from-texture cues utilizing one or more texels. Since the constraint is valid the texels must be part of the same surface patch and can be grouped together. By iterating through all of the valid constraints the texels can be grouped together to form surfaces.

The surfaces are generated by combining texels based on orientation information and texture classification assumptions. This allows a greater flexibility in the surface segmentation. Images containing surfaces with closely located texels as well as images containing transparent overlapping surfaces (see [15]) can be segmented.

4.3.4 Test Domain

There are two basic classes of shape-from-texture: texel based (normally "man-made" textures, e.g. aerial imagery of cities) and texel grouping based (normally "natural" textures, e.g. tree bark). In the texel based methods (e.g. shape-from-uniform-texel-size [16]) the textural elements are large and there is texel to texel uniformity in the unoriented textured surface. In the texel grouping based methods (e.g. shape-from-edge-isotropy [19, 6]) the uniformity in the texture can only be measured across groupings of texels (normally edge elements).

A test system has been implemented that contains three texel based methods: shape-from-uniform-texel-spacing [11], shape-from-uniform-texel-size [16], and shape-from-virtual-parallel-lines [15]. In this system the texel patches are generated by a simple histogram bin thresholding algorithm that defines each eight connected blobs as a single texel.

Shape-from-uniform-texel-spacing derives orientation constraints based on the assumption that the surface texels* can be of arbitrary shape but are equally spaced. The method takes groups of three texels and derives an orientation constraint based on the change in spacing between them (for the mathematical formulation see [15]).

---

*A surface texel is defined as the undistorted texel, as compared to, an image texel which is the distorted version appearing in the image.

The uniform spacing assumption constrains the class of textures to which the cue is applicable. Therefore, if the cue generated valid constraints, then for those texels used in generating the valid constraints the texture must be uniformly spaced. If all of the constraints in a region of the image are found to be uniformly spaced the result is similar to that recovered by a spatial frequency feature of a standard texture analysis algorithm [13].

The Second shape-from-texture method, shape-from-uniform-texel-size, utilizes the assumption that the surface texels are of uniform size, but not necessarily of uniform shape, prior to the effects of perspective distortion. This assumption is unrelated to, but not inconsistent with, the assumptions of either of the previous cues.

Given the size of two image texels and the spacing between the texels, the shape-from-uniform-texel-size cue can recover an orientation constraint (for a mathematical formulation see [15]). Since a texture is assumed to have more than two texels, this cue is able to generate sufficient constraints to solve the surface parameters.

The Third shape-from-texture method is shape-from-virtual-parallel-lines. The major assumption of this cue is that the surface texels are located on virtual parallel lines [15]. This is a related weaker assumption to that of shape-from-uniform-texel-spacing.

A virtual line is defined as the imaginary line connecting the center of mass of two or more image texels. Other virtual line definitions are possible and would generate additional orientation constraints. Since the virtual lines that connect the texels are parallel on the surface, the lack of parallelness in the image constrains the surface orientation. The point at which the lines converge is a vanishing point, which by definition is an orientation constraint.

The three cues, described above, are a useful subset of all possible shape-from-texture cues. They are able to resolve the surface parameters of a class of difficult textured surfaces (see [15]). Furthermore, as will be shown in the next section, they can segment images into surfaces and are useful in limiting the class of textures and aiding in surface segmentation.

4.3.5 Experimental Results

The system has been tested over a range of both synthetic and natural textured surfaces, and shows robustness and generality. We show here one example of real, noisy image that demonstrates the applicability of a multiple shape-from-texture system to shape recovery, texture analysis, and image segmentation.

The image (see figure 4-5) shows a camera-acquired image of a computer terminal keyboard.

The key tops are chosen by the system as texels due to the grey level disparity between them and the majority of the pixels in the image. The key top texels, shown in figure 4-5, are uniformly sized. They are also uniformly spaced horizontally, and approximately uniformly spaced vertically. Unfortunately, they are not linearly positioned in any but the horizontal direction, therefore the only constraints generated by shape-from-uniform-texel-spacing are in the horizontal dimension.

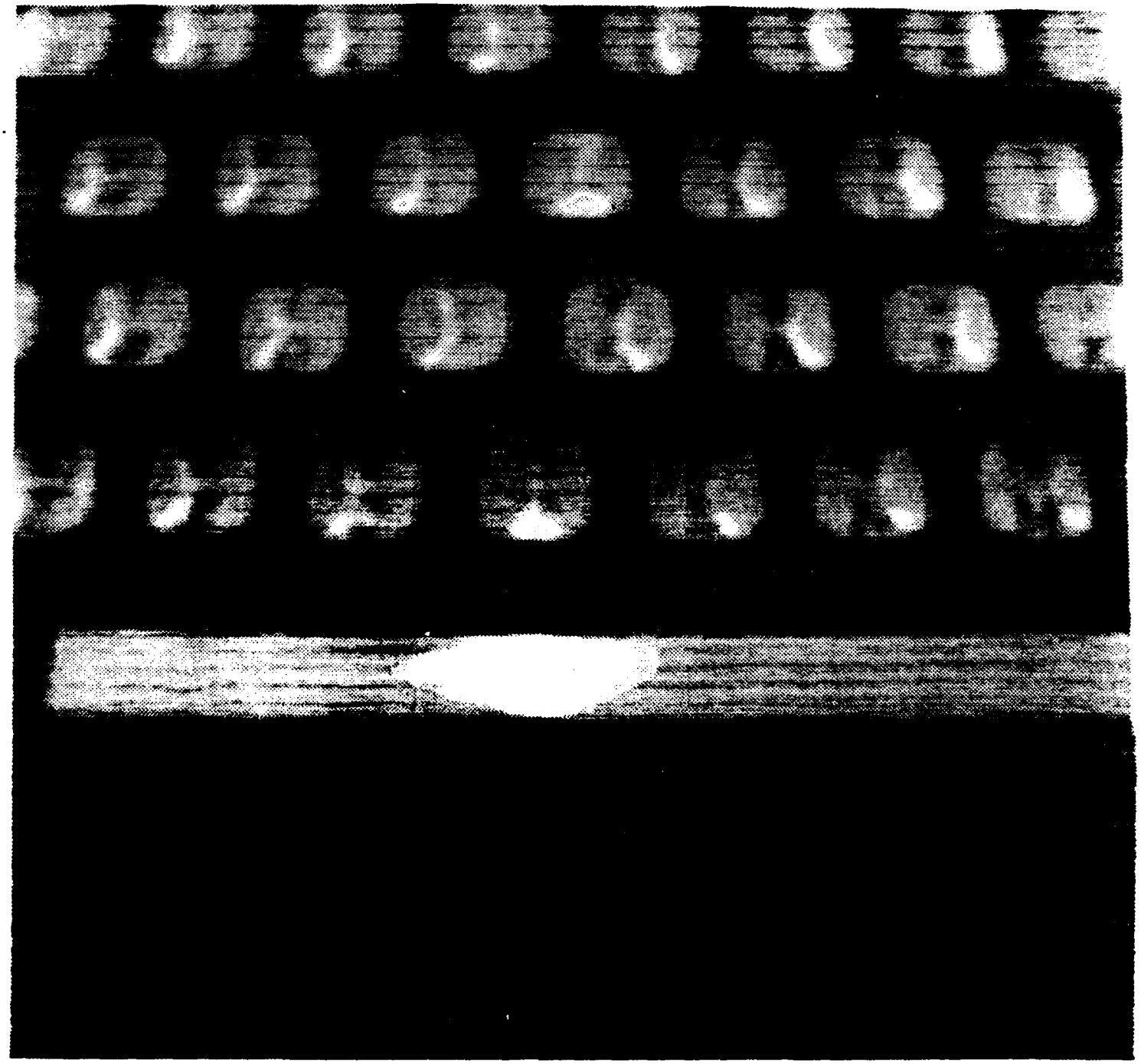


Figure 4-5: A computer terminal keyboard

The key top texels are poorly defined in the image due to digitization and camera focus errors. Furthermore, many of the key tops are inscribed with letters and have shadows cast upon them. Therefore, the threshold based blob finding algorithm has difficulty in correctly recovering texels, as shown in figure 4-6. Yet the system is able to generate, for most of the texels, the correct orientation within the measurement error (see figure 4-7)).

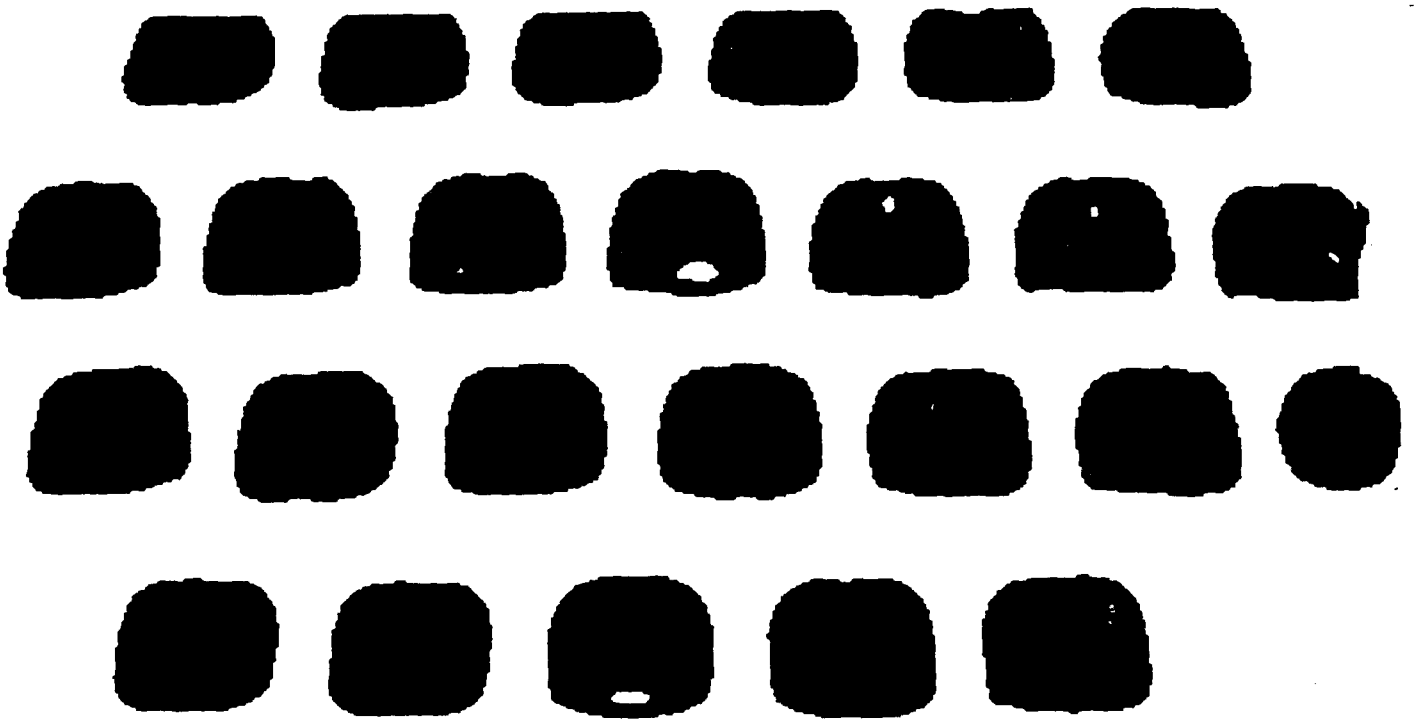


Figure 4-6: The texels of figure 4-5

Texture analysis determines that all of the texels in the image are uniformly sized, uniformly spaced horizontally, and are located on parallel horizontal lines. This information, which is easily recovered by the integrated approach, strongly constrains the texture classification problem. Furthermore, as more shape-from-texture cues (e.g. shape-from-texel-isotropy [19]) are added to the system, texture analysis will be able to generate additional texture classification information.

Surface segmentation groups the texels of figure 4-5 into four separate horizontal, groupings based on information generated by shape-from-uniform-texel-spacing and shape-from-virtual-

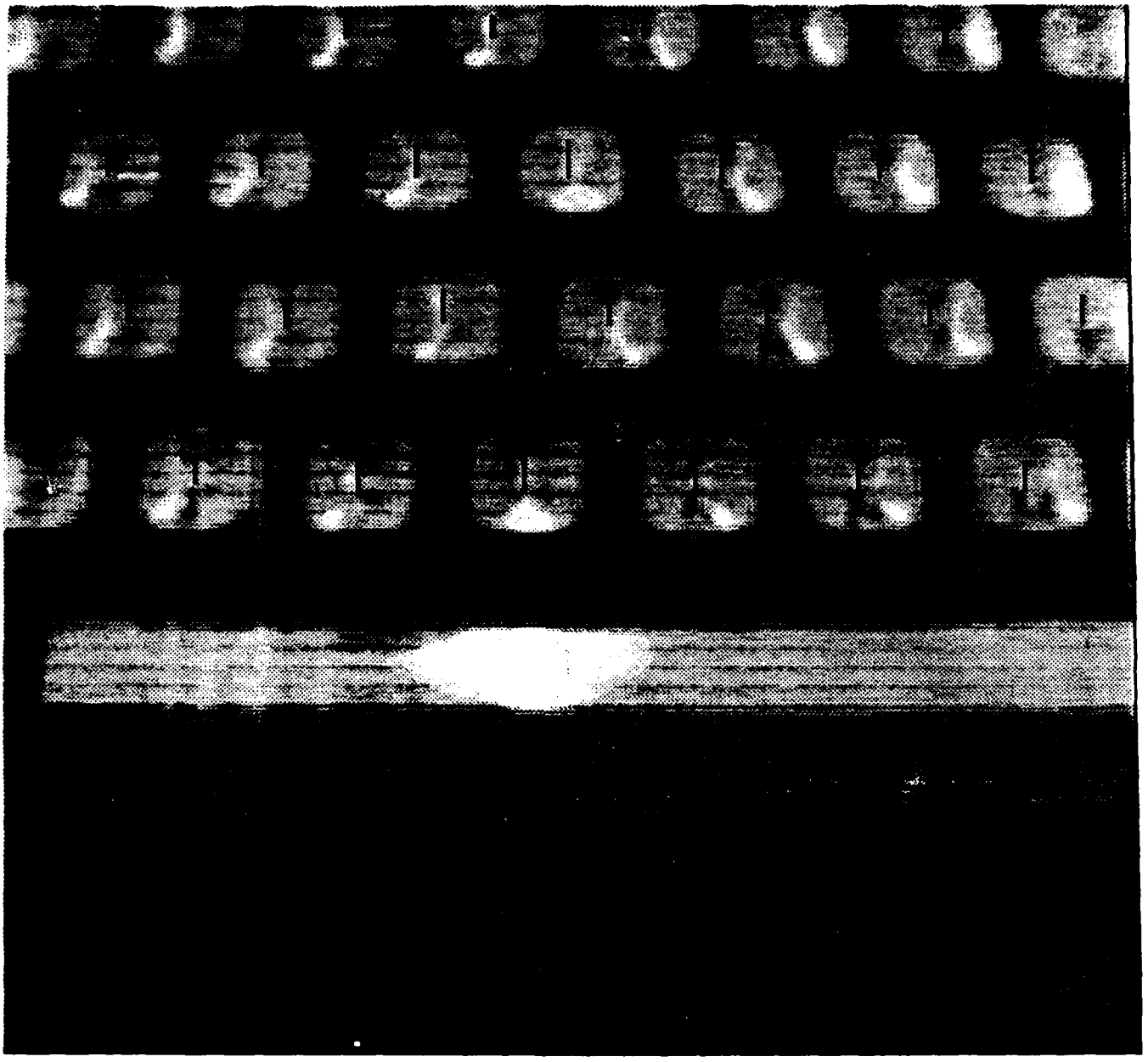


Figure 4-7: The surface orientation values for the image containing the computer keyboard

parallel-lines. Shape-from-uniform-texel-size supplies additional information that allows the horizontally grouped texels to be combined into a single surface. The system is able to recover a single surface where many purely feature-based segmentation methods, such as some spatial features methods, would fail.

4.3.6 Conclusion And Future Research

In this section we described a system that can integrate multiple shape-from-texture cues into a single system that not only generates surface shape parameters, but also performs texture analysis and surface segmentation. The system has been tested using three shape-from methods: shape-from-uniform-texel-spacing, shape-from-uniform-texel-size, and shape-from-virtual-parallel-lines and has shown the ability, under noisy conditions, to recover surface orientation, aid in texture classification and segment images into surfaces.

The segmentation algorithm does not, as yet, supply any additional information that shows exactly where the surface break should occur. This information will have to be supplied by other surface segmentation algorithms that would be integrated, in the future, with the multiple shape-from-texture system to derive a more general vision system.

The robustness of the system has been demonstrated using images that contain multiple surfaces, surfaces that are solvable by any of the methods alone, and finally with images that are solvable by using only a combination of methods.

Future enhancements to the system will include the addition of other shape-from-texture modules, the optimization of the method, especially in a parallel processing environment, and the integration of addition shape-from methods (e.g. shape-from-contour or shape-from-binocular-stereo). Other forms of texture analysis and surface segmentation will also be fused into the multiple shape-from-texture system to create a more general texture-based vision system.

5. Conclusion and Future Research

In the first stage of this project, we have developed and implemented several parallel stereo and texture vision algorithms for highly parallel computer architectures. They include a new autocorrelation-based texture algorithms, multi-resolution stereo algorithms, and depth interpolation algorithms. Initial results on the integration of stereo and texture information have been obtained, and a system for fusion of information from various shape-from-texture methods has been developed. Also, initial results of integrating stereo and texture have been demonstrated. An environment to program pyramid and multi-resolution algorithms on the Connection Machine, a highly parallel fine-grained SIMD machine, has been developed, and has been used to implement several primitives of the developed parallel algorithms.

The second stage of this project as initially proposed called for the detailed implementation of the developed algorithms on highly parallel architectures, including the newly developed textured algorithms and the system that fuses information from various shape-from-texture methods. Another continuation for this research is to develop and implement parallel algorithms for computer vision based on strong mathematical foundations such as Information-Based Complexity (IBC). They include optic flow, shape-from shading, shape-from-texture, and shape-from-stereo algorithms. In implementing these algorithms, two parallel architectures can be studied for performance; namely the Connection Machine and the WARP machine (a 10-stage pipelined high performance machine).

The implemented systems will integrate the information from stereo and texture methods to increase the certainty of computing surface parameters in the image, and they can be augmented by the use of landmarks for position location (the calculation of the global area), which can improve the performance of an autonomous land vehicle navigation system.

Other research areas is to investigate integrating information from vision and odometry cues to accurately correct the position location computation. This research will result in experiments that can be directly applied to a land vision systems such as using our new texture algorithms for determining surface slopes in a natural environment, and the use of the position location system to correct errors in odometry and other distance measurement systems.

Appendix

I. Pyramid Emulator Code on the Connection Machine

The listings which follow include all the code which is necessary to run the pyramid emulator as described in the User's Manual for Pyramid Emulation on the Connection Machine. As described in the chapter entitled "Getting Started", if you load the file "pyramid-emulate.lisp", all the other files will be loaded appropiately. Alternatively you can load "pmd-init.lisp" as a simple test. The files are included in this section in the same order as they are loaded in, which is:

```
pyramid-emulate.lisp
                    -- this file loads the pyramid emulator
                          and initializes the system

pmd-util.lisp       -- utility routines for pyramid emulator
                     (low level functions - transparent to user)

pmd-prime.lisp      -- pyramid primitives (allocating, defining,
                       and setting pyramids and their levels)

shift-news.lisp     -- intra-level pyramid communication functions

top-down.lisp       -- inter-level pyramid communciation functions

pmd-display.lisp    -- pyramid display routines

cube-grid.lisp      -- addressing scheme needed for display

pmd-conv.lisp       -- pyramidal convolution routines

make-mask.lisp      -- standard masks used for convolution

pmd-pref.lisp       -- "pref" function for pyramids (parallel
                          pyramid references)

pmd-edges.lisp      -- routines for pyramidal edge finding

pmd-load.lisp       -- loading image data from files into pyramids

pmd-unload.lisp     -- unloading pyramids into image files

pmd-init.lisp       -- file used to test a pyramid set-up

pmd-example  --  file containing example of user's session on
                 the Connection Machine showing how the emulator
                 is loaded in and used.
```

```
;-------------------------------------------------
;file:  pyramid-emulate.lisp
;by:    Cindy Norman/Lisa Brown/Qifan Ju
;date:  2/88, 5/88
;version: 1.0
;-------------------------------------------------

;-------------------------------------------------
;LOAD ALL FILES NEEDED FOR PYRAMID EMULATION
;-------------------------------------------------

;file with pyramid emulation utility routines
(load "pmd-util.lisp" :verbose nil)

;file containing pyramid primitives
(load "pmd-prime.lisp" :verbose nil)

;files with pyramid communication routines
(load "shift-news.lisp" :verbose nil)

(load "top-down.lisp" :verbose nil)

;files containing routines for pyramid display
(load "pmd-display.lisp" :verbose nil)
(load "cube-grid.lisp" :verbose nil)

;for convolution
(load "pmd-conv.lisp" :verbose nil)

;for the test of convolution, making some arrays
(load "make-mask.lisp" :verbose nil)

;function to get values from pmd
(load "pmd-pref.lisp" :verbose nil)

;find edges in pmd
(load "pmd-edges.lisp" :verbose nil)

;files for interface with image data in files
(load "pmd-load.lisp" :verbose nil)
(load "pmd-unload.lisp" :verbose nil)

;-------------------------------------------------
;main function for doing all pyramid emulation
(defun pyramid-emulate (&optional number-of-levels)
  (format t "~%~% **** Pyramid Emulation *****~%~%")
  (format t     "      -- version 1.0 ---~%~%")
  (cond ((test-for-pyramid-configuration)
         (cond ((eq number-of-levels nil)
```

```
                (setq *pmd-number-of-levels*
                      (+ 1
                (/ *log-number-of-processors-limit* 2))))
                 (t (cond ((= number-of-levels
                      (+ 1
                 (/ *log-number-of-processors-limit* 2)))
           (setq *pmd-number-of-levels* number-of-levels))
            (t (format t
            (*** error: improper number-of-levels~%"))))))
    (t (format t "*** error: improper configuration ~%")))

; Define other global variables needed by pyramid
; emulation

;size of one side of base of pyramid
    (setq *pmd-size* (dimension-size 0))
;mapping of hypercube connections to addressing scheme
    (*defvar *pmd-self-address* (self-address!!))
;each processor gets its non-zero level number
    (*defvar *pmd-level-number* (assign-levels!!))
;lists needed for pyramid communications
    (setq *pmd-north-south-list*
                (build-north-south-list *pmd-size*))
    (setq *pmd-east-west-list*
                 (build-east-west-list *pmd-size*))
;list of all pyramids created
    (setq *pmd-names* nil)

     t

);end pyramid-emulate
```

```
;-------------------------------------------------------------
;file:  pmd-util.lisp
;-------------------------------------------------------------

;returns pvar which is T when the value is a whole number
;(defun intgp!! (pvar)
;  (cond!! ((=!! pvar (round!! pvar)) t!!) (t!! nil!!)))

;this procedure figures out to what power of 2 the number is.
;e.g. if given 16, it will send back 4.
(defun power-of-2 (num)
 (cond ((= (log num 2) (round (log num 2))) (round (log num 2)))
       ;num ok
         (t (print "size is not a power of 2") ;num not ok
                'nil)))

;this function will send back the i(th) bit of address
;address is parallel, i is not.
(*defun *get-bit  ( i paddress)
       (cond!! (   (=!! (!! 0)
                        (logand!! paddress (!! (expt 2 (- i 1)))))
     ;if AND of paddress and 2**(i-1) = 0, then bit is 0 else 1
                     (!! 0))
                (t!! (!! 1))))

;this just subs 1 from even numbers and leaves alone if odd
(defun set-odd (num)
       (cond (  (evenp num) (- num 1))
             (t num)))

;this just adds 1 to odd numbers and leaves alone if even
(defun set-even (num)
       (cond (  (oddp num) (+ num 1))
             (t num)))

;used by assign-levels!!
(defun intgp!! (pvar)
  (*let (( tpvar nil!!))
        (*when (=!! pvar (!! 0))
               (*set tpvar t!!))
        (*when (/=!! pvar (!! 0))
               (*set tpvar nil!!))
        tpvar))

;set each processor to the number of the level of the pyramid
;that it emulates (since all emulate zero, assign it zero
;only when that is the only level it emulates.)
(*defun *set-level (pvar level)
  (*when (intgp!! (mod!! (+!! *pmd-self-address*
```

```
                                (!! (expt 2 (- level 1))) )
                            (!! (expt 4 level))))
          (*set pvar (!! level))))

(*defun  assign-levels!! ()
  (*all
   (*let ((plevel (!! 0)))
    (do
      ( (x (- *pmd-number-of-levels* 1) (- x 1)))
      ( (= x 0) t)
      (*when (=!! plevel (!! 0))
             (*set-level plevel x)))
    plevel)))


; this function will create the entire list of dimensions
; needed to travel down for a shift to
; the west or east (odd numbers). This list is used for
; all levels. Each level will stop at different points on
; the list.
(defun build-east-west-list (size)
  ;size is # pixels on side of image
 (build-sel2 (set-odd (round (power-of-2 (* size size)))) nil))


; this is the same as build-east-west-list but builds the list
; of even ;numbers-the numbers needed for north-south shifts.
(defun build-north-south-list (size)
 ;size is # pixels on side of image
 (build-sel2 (set-even (round (power-of-2 (* size size)))) nil))

; this function actually builds the list of send-east
; commands described on the top of page 6 of Hussein's paper.
; this also will build the send-north and send-south lists -
; depending upon the parameters. for these two, you need to
; send an even number for high-num and it will build a list
; of even numbers stopping at 2-which is the stopping
; point for north/south sends (level 0). otherwise, it
; will stop at 1 which is the stopping point for east/west
; sends (level 0) Depending upon what level you are working
; with will depend upon how much of this list you will need.
; If level 0, you will need the whole list, if ;level 1/send
; east-west, you will need all but the '1' in the list. if
; north/south sends, you will need all but the '2' in the list.
(defun build-sel2 (high-num lis)
   (cond (  (< high-num 1 ) lis)
  ;finished building list-STOP AT LEVEL 1
  (t (build-sel2 (- high-num 2) (append lis (list high-num) )))))
```

```
; this function test that the current CM configuration is
; appropriate for pyramid emulation
(defun test-for-pyramid-configuration ()
; note: need to reconsider these conditions
   (cond ((and (integerp *log-number-of-processors-limit*)
               (= *number-of-dimensions* 2))
          (cond ((eq (dimension-size 0) (dimension-size 1))
                 t)
                (t nil)))
          (t nil)))
```

```
;-------------------------------------------------
; file : pmd-prime.lisp
;-------------------------------------------------

; define pmd as a new structure which has two
; pvars: top and leaf
(*defstruct (pmd :immediate-data-type t)
       (top   (!! 0) :type (signed-pvar 100))
       (leaf  (!! 0) :type (signed-pvar 100)))

(*defun allocate-pmd!! ()
    (make-pmd!!))

(*defun *deallocate-pmd (pmd)
     (*deallocate pmd)

  )
; define a new named pmd . if it exists, reset it.
(defmacro *defpmd (pmdname &optional1 init-pmd)
   `(let ((newname (quote ,pmdname)))
      (cond ((not (quote ,init-pmd ) )
             (*defvar ,pmdname (make-pmd!!))
             newname)
            (t (*defvar ,pmdname (make-pmd!!))
               (*set-pmdstruct ,pmdname ,init-pmd)
               newname))))


; set the value in pmd2 into pmd1
(defun *set-pmd (pmd-1 pmd-2)
  (cond ((and (pvarp pmd-1) (pvarp pmd-2))
         (cond ((and (pmdp pmd-1) (pmdp pmd-2))
                (*set-pmdstruct pmd-1 pmd-2))
               (t  (format t "~% not a pmd structure ~%"))))
        (t   (format t "~% not a pmd structure ~%"))))


; assign pmd-2 into pmd-1
(*defun *set-pmdstruct (pmd-1 pmd-2)
  (*all
    (setf (pmd-top!! pmd-1) (pmd-top!! pmd-2))
    (setf (pmd-leaf!! pmd-1) (pmd-leaf!! pmd-2))))

(*defun *set-pmd-let (pmd-1 pmd-2)
  (*all
    (setf (pmd-top!! pmd-1) (pmd-top!! pmd-2))
    (setf (pmd-leaf!! pmd-1) (pmd-leaf!! pmd-2))))
```

```
; return a pmd structure. the value of each processor is num.
(*defun pmd!! (num)
   (make-pmd!! :top (!! num) :leaf (!! num)))

; assign value of the pvar to the top and the leaf of
; pmd1 structure.
(*defun *set-pmd-pvar (pmd1 pvar)
    (setf (pmd-top!! pmd1)  pvar)
    (setf (pmd-leaf!! pmd1) pvar))


; assign the values of specific level from pmd-2 to pmd1.
(*defun *set-level-pmd (level pmd-1 pmd-2)
  (*all
    (cond ((= level 0)
           (setf (pmd-leaf!! pmd-1) (pmd-leaf!! pmd-2)))
          (t  (setf (pmd-top!! pmd-1)
            (cond!! ((=!! (!! level) *pmd-level-number*)
                          (pmd-top!! pmd-2))
                         (t!! (pmd-top!! pmd-1)))))))))


; assign value of the pvar to a specific level of pmd1
; structure.
(*defun *set-level-pmd-pvar (level pmd-1 pvar)
  (*all
    (cond ((= level 0)
           (setf (pmd-leaf!! pmd-1) pvar))
          (t  (setf (pmd-top!! pmd-1)
            (cond!! ((=!! (!! level) *pmd-level-number*)
                          pvar)
                         (t!! (pmd-top!! pmd-1)))))))))



; get the value when given cube address.
; top=1 means top, top=0 means leaf.
(*defun pref-cube-pmd (top pmd num)
   (cond ((=  top 1)
          (pref-pmd-top pmd num))
         (t  (pref-pmd-leaf pmd num))))


;print part of a pvar.
(*defun *ppp-part (pvar size)
         (dotimes (x size)
           (dotimes (y size)
               (format T "~6d "
                  (pref-grid pvar x y)))
```

```
            (terpri)))

;; pmd structure predicator.
(*defun pmdp (pmd)
   (cond ((*and (structurep!! pmd))
          t)
         (t nil)))

; return a pvar from a pmd according to the level number.
(*defun pmd2pvar!! (pmd level)
    (cond ((= level 0)
           (pmd-leaf!! pmd))
          (t (pmd-top!! pmd))))
```

```
;-------------------------------------------------------------
;file: shift-news.lisp
;date: 2/88, 5/88
;-------------------------------------------------------------

;-------------------------------------------------------------
;This is a support function for the *shift operations below.
;<lis> is a list of dimensions (see build routines in
; pmd-util.lisp)
;-------------------------------------------------------------
(*defun *send-east-south (lis level source-pvar dest-pvar)
 ; stop sending after sending forward along the
 ;  2*level+1 dimension
        (*when  (or!! (>=!! (!! (car lis))
                      (1+!! (*!! (!! 2)  *pmd-level-number*)))
                         (=!! (!! level) (!! 0)))
                (cond  (    (eq nil (cdr lis))
                       (*send-forward (car lis)
                                      source-pvar dest-pvar))
                       (t    (*send-forward (car lis)
                                            source-pvar dest-pvar)
                          (*when (or!! (>=!! (!! (cadr lis))
                                             (1+!! (*!! (!! 2)
                                                *pmd-level-number*)))
                                       (=!! (!! level) (!! 0)))
                                 (*send-backward (cadr lis)
                                                 dest-pvar dest-pvar))
                           (*send-east-south (cdr lis) level
                                             source-pvar
                                             dest-pvar))) )
);end *send-east-south


;this is similar to *send-east-south but in these you have the
;pattern of  sending starting with send-back then send-forward
;send-back until the end of <lis>.
(*defun *send-west-north (lis level source-pvar dest-pvar)
        (*when (or!! (>=!! (!! (car lis))
                     (1+!! (*!! (!! 2)  *pmd-level-number*)))
                      (=!! (!! level) (!! 0)))
               (cond  (    (eq nil (cdr lis))
                           (*send-backward (car lis) source-pvar
                                           dest-pvar))
                      (t   (*send-backward (car lis) source-pvar
                                           dest-pvar)
                           (*when (or!! (>=!! (!! (cadr lis))
                                              (1+!! (*!! (!! 2)
                                                 *pmd-level-number*)))
                                        (=!! (!! level) (!! 0)))
```

```
                                        (*send-forward (cadr lis)
                                                       dest-pvar dest-pvar))
                        (*send-west-north (cdr lis)  level source-pvar
                                                     dest-pvar))) )
);end *send-west-north

;send data forward along the dimension <dim> from the
;<source-pvar> to the <dest-pvar>
(*defun *send-forward  (dim source-pvar dest-pvar)
        ;special case when <dim>=0 (dest=source)
;(format t "forward on ~2d" dim)
        (cond (    (= dim 0)
                   (*all (*set dest-pvar source-pvar)) )
   ;send data forward only if <dim>th bit of address is zero
        (t  (*when (=!! (!! 0) (*get-bit dim *pmd-self-address*))
                ;data received in processor with same address
                ;but <dim>th bit is one
             (setf (pref!! dest-pvar (+!! (!! (expt 2 (- dim 1)))
                                          *pmd-self-address*))
                           (pref!! source-pvar *pmd-self-address*))))
         );end cond
);end *send-forward

;-------------------------------------------------------------
; this works the same as *send-forward except an operation is
; performed on the data being sent and the data at the
; address of the  destination (both from source-pvar) before
; it is placed into the dest-pvar.
;-------------------------------------------------------------
(*defun *send-forward-op (dim source-pvar dest-pvar op)
     ;send data forward only if <dim>th bit of address is zero
      (*when (=!! (!! 0) (*get-bit dim *pmd-self-address*))
           (setf (pref!! dest-pvar (+!! (!! (expt 2 (- dim 1)))
                                        *pmd-self-address*))
     ;this calls the function sent in <op> & applies to psource
             ;of first address and psource of second address.
            (*funcall op (pref!! source-pvar *pmd-self-address*)
                                 (pref!! source-pvar (+!!
                                        (!! (expt 2 (- dim 1)))
                                          *pmd-self-address*)))
                 );end setf
        );end when
);end *send-forward

;-------------------------------------------------------------
; this works the same as *send-forward except that the
; dimension  <dim> is travelled 'back' on. The data in
; source-pvar whose <dim>th bit of its address is 1 is sent to
; dest-pvar at the corresponding address whose <dim>th bit is 0.
```

```
;-------------------------------------------------------------
(*defun *send-backward (dim source-pvar dest-pvar)
;(format t "backward on ~2d" dim)
        (cond (    (= dim 0)
                   (*all (*set dest-pvar source-pvar)) )
        (t   (*when (=!! (!! 1) (*get-bit dim *pmd-self-address*))
                    (setf (pref!! dest-pvar (-!! *pmd-self-address*
                                                 (!! (expt 2 (- dim 1)))))
                          (pref!! source-pvar *pmd-self-address*))) )
         );end cond
);end *send-backward


(*defun *shift (direction level source-pvar dest-pvar)
   (cond  (  (eq direction  'e)
                ;use east-west-list :the odd numbers
             (*send-east-south *pmd-east-west-list* level
                               source-pvar dest-pvar))
          (  (eq direction  'w) ;
             (*send-west-north *pmd-east-west-list* level
                               source-pvar dest-pvar))
          (  (eq direction  'n)
              ;use north-south-list : the even numbers
           (*send-west-north *pmd-north-south-list* level
                               source-pvar dest-pvar))
          (t ;(eq direction 's)
           (*send-east-south *pmd-north-south-list* level
                             source-pvar dest-pvar)) )
);end *shift

;-------------------------------------------------------------
; this function will only shift a particular level.
; the 'all is used because if some other function had
; a subset of processors set before this one is called, all
; of the processors may  not be checked for the *when clause.
;-------------------------------------------------------------

(*defun shift-level-pmd!! (level direction source-pmd
                                &optional dest-pmd
                                &key border-pmd)
    (*let ( (dest-pvar-temp  (!! 0))
            (source-pvar (pmd2pvar!! source-pmd level)) )

        (cond ((not border-pmd)
                (*all (*set dest-pvar-temp (!! 0))))
              (t (*all (*set dest-pvar-temp
                             (pmd2pvar!! border-pmd level)))))

        (cond ( (eq level 0)
```

```
                ; level 0 then all processors enabled
   (*all (*shift direction level source-pvar dest-pvar-temp)) )
     ( t              ; other levels, enable relevant processors
              (*all (*when (=!! *pmd-level-number* (!! level))
                           (*shift direction level source-pvar
                                   dest-pvar-temp) )))
              );end condition

        ;return temp dest-pvar as part of pmd (rest unchanged)
      (cond (  (not dest-pmd)
               (cond ((= level 0)
                      (make-pmd!!  :top (pmd-top!! source-pmd)
                                   :leaf dest-pvar-temp))
                     (t (make-pmd!! :top dest-pvar-temp
                          :leaf (pmd-leaf!! source-pmd)))) )
      (t
      (cond ((= level 0)
             (*all (setf (pmd-leaf!! dest-pmd) dest-pvar-temp)
                         (setf (pmd-top!!  dest-pmd)
                               (pmd-top!! source-pmd)))
                   dest-pmd)
                (t  (*all (*set-pmd-let dest-pmd source-pmd))
                    (*when (=!! *pmd-level-number* (!! level))
                           (*all (setf  (pmd-top!! dest-pmd)
                                        dest-pvar-temp)))
                       dest-pmd))
              ))

      );end let
);end shift-level-pmd!!

;-------------------------------------------------------------
;this function will shift all levels in the specified <direction>
;-------------------------------------------------------------
(*defun shift-pmd!! (direction  source-pmd &optional dest-pmd
                                                 &key border-pmd)
    (*let ( (source-pvar (pmd-top!! source-pmd))
            (dest-pvar (!! 0))
            (temp-pmd (make-pmd!!)))

          (*all (*when (>!! *pmd-level-number* (!! 0))
                       (*shift direction 1 source-pvar
                               dest-pvar)
                       (cond (border-pmd
                              (*when (=!! dest-pvar (!! 0))
                                     (*set dest-pvar
                                      (pmd-top!! border-pmd))))))
```

```
            (*all (shift-level-pmd!! 0 direction source-pmd temp-pmd
                                     :border-pmd border-pmd))

            (cond (dest-pmd
                   (setf (pmd-top!! dest-pmd) dest-pvar)
                  (setf (pmd-leaf!! dest-pmd) (pmd-leaf!! temp-pmd)))
                  (t
                   (setf (pmd-top!! temp-pmd) dest-pvar)
                   dest-pmd))

))));end shift-pmd!!
```

```
;-------------------------------------------------------------
;file: top-down.lisp
;version: 2.0
;-------------------------------------------------

;-------------------------------------------------
; THIS FILE CONTAINS WHAT IS NEEDED TO SEND TOP/DOWN
; COMMUNICATIONS IN THE HYPERCUBE FORMAT. SOME OTHER
; SUPPORT FUNCTIONS ARE NEEDED.THEY MAY BE FOUND
; IN SHIFT-NEWS.LISP  or PMD-util.LISP
; (e.g. (*assign-levels)(reset-pvars)(shift-all!!...)
; (print-image pvar)...
; (*send-forward dim psource pdest)(*send-backward
; <same as -forward> )
;
;THE TWO MAIN FUNCTIONS:
;(send-level-parent-pmd!!   child-level
;                  op-to-perform-on-source-pvar-and-dest-pvar
;                       source-pvar-on-child-level
; &optional              dest-pvar-on-parent-level)
;
;(send-level-child-pmd!! child-level
;                         child-a-b-c-or-d
;                         source-pvar-on-child-level
;&optional                dest-pvar-on-parent-level)
;
;(send-level-children-pmd!! parent-level
;                        source-pvar-from-parent-level
;                        dest-pvar-of-children-level)
;-------------------------------------------------------------

;-------------------------------------------------------------
; this will send info from level i to i+1 -- the parent. It
; first travels  along the 2i+1(th) dimension. so, e.g. if
; we were going from level 0 to  level 1, 0's source-pvar
; would go to 1's dest-pvar(pdest99) and 2's source-pvar
; would go to 3's dest-pvar. When these pvars get sent to
; the next dim, they are op(d) with the source-pvar of the
; resultant  address-the value is put into the resultant
; dest-pvar. e.g. 0's  source-pvar gets op(ed) with 1's
; source-pvar and the result gets put into 1's dest-pvar.
; next the 2i+2 dimension's is travelled on ( 1 goes to 3).
; third, backward on i (3 goes to 3 - esentially nothing happens
; for level 0) THE METHOD: send forward along dimensions 2i+1
; then 2i+2 then backwards on  dimension i.
;  if you can't send along 2i+1 or 2i+2 then you skip that
; step-must do (and always can do except when i=0) the backwards
; on i. this works for all levels.
; 'pdest99 is a temporary pvar needed when shifting information
```

```
; around. At the end of the *let, this pvar is referenced so
; that its value will be sent back to the calling function.
; This will enable the user to assign the value to whatever
; pvar she wishes.
;---------------------------------------------------------------
(*defun send-level-parent-pmd!! (level op source-pmd
                                          &optional dest-pmd)

;       if able to send to level above-e.g. if only 3 levels,
;       can't send to level 4

    (cond ( (> level (- *pmd-number-of-levels* 2))
         (format t  "~3d is too high of a level to use." level))
         (t
          (*let  ( (pdest99 (!! 0))  ;only temporary pvar needed
                   (source-pvar (pmd2pvar!! source-pmd level)) )
               (*all
                (*when (or!! (=!! *pmd-level-number* (!! level))
                             (=!! (!! level) (!! 0)))
                      (*send-forward-op
                                      (+ (*  2 level) 1)
                                      source-pvar
                                      pdest99
                                      op)

                      (*send-forward-op
                                      (+ (* 2 level)  2)
                                      pdest99
                                      pdest99
                                      op)
                      (*send-backward
           ;back on i-don't use op because just sending #
                                     level
                                     pdest99
                                     pdest99)

                 ));end  when  and all

                 ;0 out all pdest99 except i+1 level
          (*all (*when (/=!! *pmd-level-number* (!! (+ level 1)))
                           (*set pdest99 (!! 0)) ))

;            Now, send back value for user to assign to own pvar
         (*all (*when (=!! *pmd-level-number* (!! (+ level 1)))
                   (cond (  (not dest-pmd)
                            (pref!! pdest99 *pmd-self-address*))
                   (t (*set-level-pmd-pvar (+ 1 level) dest-pmd
                                     pdest99)))))
))));end *let/true/cond
```

```
;end send-parent!!


;-------------------------------------------------------------
; send the average value of child to parent level.
;-------------------------------------------------------------
(*defun send-level-parent-average-pmd!! ( level source-pmd
                                          &optional dest-pmd)
 (*all
    (*let ((temp-pmd (make-pmd!!))
           (temp-pvar (!! 0)))

          (*all (*set temp-pvar (!! 0)))
          (send-level-parent-pmd!! level '+!!
                                   source-pmd
                                   temp-pmd)

          (*when (=!! *pmd-level-number* (!! (+ level 1)))
                 (*all (*set-level-pmd-pvar
                  (+ level 1)
                 temp-pmd
                 (round!! (/!! (pmd-top!! temp-pmd) (!! 4))))))

          (cond ((not dest-pmd)
                 temp-pmd)
             (t (*set-level-pmd (+ level 1) dest-pmd temp-pmd)
                   dest-pmd))
)))


(*defun average-pmd!! (pvar &optional dest-pmd)
 (*all
   (*let  ((temp-pmd (make-pmd!!)))
          (setf (pmd-leaf!! temp-pmd) pvar)
          (setf (pmd-top!! temp-pmd) pvar)
          (dotimes (level (- *pmd-number-of-levels* 1))
                   (send-level-parent-average-pmd!!
                     level temp-pmd temp-pmd))
          (cond ((not dest-pmd)
                 temp-pmd)
                (t (*set-pmd dest-pmd temp-pmd)
                   dest-pmd)))))

;-------------------------------------------------------------
; send  a particular child to parent level
;-------------------------------------------------------------
(*defun send-level-child-pmd!! (level child source-pmd
                                        &optional dest-pmd)
```

```
        (*let  ( (dest-pvar (!! 0))
                 (source-pvar (pmd2pvar!! source-pmd level))
                 (temp-pmd (make-pmd!!)) )
               (*all
                (*when (or!! (=!! *pmd-level-number* (!! level))
                             (=!! (!! level) (!! 0)))
                       (send-part-child2!! child source-pvar
                                           dest-pvar level)) )

;        Now, send back value for user to assign to own pvar
        (*all (*when (=!! *pmd-level-number* (!! (+ level 1)))
                     (cond ((not dest-pmd)
                            (*set-level-pmd-pvar
                              (+ level 1)
                              temp-pmd
                              dest-pvar)
                            temp-pmd)
                           (t (*set-level-pmd-pvar
                                (+ 1 level)
                                dest-pmd
                                dest-pvar)))))
));end send-level-child-pmd!!



;-------------------------------------------------------------
; This function sends a particular child to its parent.
; It is called from the function:send-particular-child!!.
; All of the children need to be sent backward along the i(th)
; dimension after the following steps are taken:
;      child 'a' needs to send-forward on each dimension.
;      child 'b' needs to send along the 2i+2.
;      child 'c' needs to send along the 2i+1 dimension.
;      child 'd' doesn't nee` any subsequent send.
;-------------------------------------------------------------
(*defun send-part-child2!! (child source dest level)
; if the child was b or d, then dest needs to be set to source
; so that the send functions will work for each child
; generically.
; (otherwise the sendforward would go from dest to source and
; dest=nil
      (if (member child '(b d))
          (*set dest source))

; if the child is 'a or 'c, then need to send along the
; 2i+1 dimension no other child needs to travel this time.
      (if (member child '(a c))
          (*send-forward  (+ (* 2 level) 1) source dest) )
```

```
      (if (member child '(a b))
          (*send-forward  (+ (* 2 level) 2) dest dest))

      (*send-backward level dest dest)
          ;send back no matter which child

);end send-part-child2!!



;-------------------------------------------------------------
; used for adding operation for send-level-parent
; this is just an op to be used when sending parallel along
; a dimension. This,  obviously, adds two pvars.
;-------------------------------------------------------------
(*defun *add2 (px py)
        (+!! px py))



;-------------------------------------------------------------
; from i+1 to i
; this will send the value in the parent processor 'level'
; :(level i+1) to all of ;  its children (level i)
; the parameters: level = parent level sending from
;       source-pvar = pvar in parent level to send down
;-------------------------------------------------------------
(*defun send-level-children-pmd!! (level source-pmd
          &optional dest-pmd)
     (*all
        (*when (or!! (=!! *pmd-level-number* (!! level))
                          ;turn on parent level to send from
                     ;turn on level to send to - children
                     (=!! *pmd-level-number* (!! (- level 1)))
                     (=!! (!! level) (!! 1))  );end 'or

            (*let ( (pdest99 (!! 0)) ;only temp pvar to use
                    (source-pvar (pmd2pvar!! source-pmd level)) )
               (*send-forward
                  (- level 1) ;forward i(level-1) from i+1(level)
                              source-pvar
                              pdest99)
               (*send-backward
                              (+ (*  2 (- level 1)) 2) ;2i+2
                              pdest99
                              pdest99)
               (*send-backward
                              (+ 1 (* 2 (- level 1))) ;2i+1
                              pdest99
```

```
                                    pdest99)
        (*all
            (*when (or!! (=!! (!! level) (!! 1))
                         (=!! *pmd-level-number* (!! (- level 1))))
                   (cond (  (not dest-pmd)
                            (pref!! pdest99 *pmd-self-address*))
                   (t (*set-level-pmd-pvar (- level 1) dest-pmd
                                           pdest99)))))

))));end send-level-children!!
```

```
;-------------------------------------------------------------
; file: pmd-display.lisp
; version: 2.0
;-------------------------------------------------------------

;-------------------------------------------------------------
; routine for display values of a certain level
; (note: uses routines in cube-grid.lisp)
;
; this function returns a pvar which is set to the x-coordinate
;  of the grid-address for the PE on the specified level - this
; pvar should be indexed by its grid-address which represents
; the x,y coordinates of the node on the level
;-------------------------------------------------------------
(*defun level-to-grid-x!! (ilevel)
       -!! (-!! (*!! (+!! (!! 1) (self-address-grid!! (!! 1)))
                          (!! (expt 2 ilevel)))
                     (!! 1))
             (cube2col!! (!! (- (expt 2 (- ilevel 1)) 1)))))

;-------------------------------------------------------------
; this function returns a pvar which is set to the y-coordinate
; of the grid-address for the PE on the specified level - this
; pvar should be indexed by its grid-address which represents
; the x,y coordinates of the node on the level
;-------------------------------------------------------------
(*defun level-to-grid-y!! (ilevel)
     (-!! (-!! (*!! (+!! (!! 1) (self-address-grid!! (!! 0)))
                          (!! (expt 2 ilevel)))
                     (!! 1))
              (cube2row!! (!! (- (expt 2 (- ilevel 1)) 1)))))

;-------------------------------------------------------------
; returns pvar which when grid-addressed by x,y will return
; the cube-address of the PE whose location on the specified
; level is x,y
;-------------------------------------------------------------
(*defun level-cube!! (ilevel)
  (cond ((= ilevel 0)
         (rowcol2cube!! (self-address-grid!! (!! 0))
                        (self-address-grid!! (!! 1))))
        (t   (rowcol2cube!! (level-to-grid-y!! ilevel)
                            (level-to-grid-x!! ilevel)))))

;-------------------------------------------------------------
; display the specified values for the specified level in grid
; format.
;-------------------------------------------------------------
(*defun *display-level-pmd (pmd ilevel)
```

```
  (*all
   (let ((level-gridsize (expt 2 (- *pmd-number-of-levels*
                                    (1+ ilevel)))))
     (*let ((plevel-cube (level-cube!! ilevel))
         (p-value (pmd2pvar!! pmd ilevel)))
         (terpri)
; for each pe on this level (in raster scan order)
         (dotimes (x level-gridsize)
           (dotimes (y level-gridsize)
               (format T "~6d "
; given "level" grid-address - reference p-value
; by first determining its cube-address
                   (pref p-value (pref-grid plevel-cube x y))))
            (terpri))))))

;-----------------------------------------------------------
;display all levels of a pyramid
;-----------------------------------------------------------
(*defun *display-pmd (pmd)
  (dotimes (ilevel *pmd-number-of-levels*)
     (format t "~% Level ~2d : ~%" ilevel)
     (*display-level-pmd pmd ilevel)))


(*defun *display-pmd-part (pmd start-level end-level)
  (do ((ilevel start-level (+ ilevel 1)))
      ((= ilevel (+ end-level 1)))
     (format t "~% Level ~2d : ~%" ilevel)
     (cond ((> ilevel 3)
            (*display-level-pmd pmd ilevel))
           (t (*display-level-pmd-part pmd ilevel 10)))))



(*defun *display-level-pmd-part (pmd ilevel size)
  (*all
   (let ((level-gridsize (expt 2
                   (- *pmd-number-of-levels* (1+ ilevel)))))
     (*let ((plevel-cube (level-cube!! ilevel))
         (p-value (pmd2pvar!! pmd ilevel)))
         (terpri)
         (cond ((< ilevel 4)
                (setq level-gridsize size)))
         (dotimes (x level-gridsize)
          (dotimes (y level-gridsize)
               (format T "~6d "
                (pref p-value (pref-grid plevel-cube x y))))
            (terpri))))))
```

```
;-------------------------------------------------------------
; file: cube-grid.lsp
;
; parallel routines for converting cube addresses to row and
; column and vice versa
;-------------------------------------------------------------

(*defun cube2row!! (pcube)
  (*let ((prow (!! 0)))
    (do ((i 15 (- i 2))) ((= i -1) prow)
        (*set prow (deposit-byte!! prow (!! (/ (- i 1) 2)) (!! 1)
                (load-byte!! pcube (!! i) (!! 1)))))))


(*defun cube2col!! (pcube)
  (*let ((pcol (!! 0)))
    (do ((i 14 (- i 2))) ((= i -2) pcol)
        (*set pcol (deposit-byte!! pcol (!! (/ i 2)) (!! 1)
                (load-byte!! pcube (!! i) (!! 1)))))))


(*defun row2cube!! (prow)
  (*let ((pcube (!! 0)))
    (do ((i 15 (- i 2))) ((= i -1) pcube)
      (*set pcube (deposit-byte!! pcube (!! i) (!! 1)
            (load-byte!! prow (!! (/ (- i 1) 2)) (!! 1)))))))

(*defun col2cube!! (pcol)
  (*let ((pcube (!! 0)))
    (do ((i 14 (- i 2))) ((= i -2) pcube)
      (*set pcube (deposit-byte!! pcube (!! i) (!! 1)
                (load-byte!! pcol (!! (/ i 2)) (!! 1)))))))

(*defun rowcol2cube!! (prow pcol)
  (+!! (row2cube!! prow) (col2cube!! pcol)))
```

```
;-------------------------------------------------------------
;  file  :   pmd-conv.lisp
;
;  functions:  to do the convolution for pmd.
;               1. for a special level
;               2. for all levels together
;
;-------------------------------------------------------------
;-------------------------------------------------------------
; this function will do the convolution for a user's specified
; level of a pyramid using a given mask.
;
; input :  pmd-   a parymid
;          level- a number to indicate the level user want
;          mask-  the mask for convolution
;          m, n-  the size of the mask
;
; return:  the user specified level of the input pmd get the
;          result of  the convolution with the given mask
;
; this function uses the functions in shift-news.lisp file.
; according to the mask's size, it do the convolution in
; the order: downward, rightward, upward, and leftward as
; a cycle. it does cycle by cycle until finish the number
; of steps , which is m*n.
;
;-------------------------------------------------------------

(*defun pmd-conv-level!! (pmd  level mask m n
                                          &optional dest-pmd)
  (*all
    (*let ((pmd-temp (make-pmd!!))
           (pmd-copy (make-pmd!!)))

      (let ((sum (sumarray mask m n ))
            (done (* m n)))

(*set-level-pmd-pvar level pmd-temp (*!! (pmd2pvar!! pmd level)
                                    (!! (aref mask 0 0))))

    (*compute-conv pmd-temp pmd-copy pmd level mask m n done )

        ) ;end of let
      (cond ((not dest-pmd)
             pmd-temp)
            (t
              (*set-level-pmd level dest-pmd pmd-temp)
             dest-pmd))
      ) ;end *let.
```

```
));end of the function.

;-------------------------------------------------------------
; it returns the sum of an array
; input:    arr is an array
;           m,n are the size of the array
;-------------------------------------------------------------
(defun sumarray(arr m n)
  (let ((sum 0))
    (do ((temp1 0 (+ temp1 1)))
        ((= temp1  m ) sum)
      (do ((temp2 0 (+ temp2 1)))
          ((= temp2  n ) )
        (setq sum (+ sum (aref arr temp1 temp2)))))))



;-------------------------------------------------------------
;gets the convolution value for a pmd
;-------------------------------------------------------------
(*defun *compute-conv (pmd-temp pmd-copy pmd level mask m n done)
 (let  ((xstart 0)
        (xend (- m 1))
        (ystart 0 )
        (yend (- n 1)))

  (do ((count 1))                              ;for one circle.
      ((= count done))

    (do ((x xstart)                        ;for downward, shift south
         (y ystart (+ y 1)))
        ((or (= y  yend ) (= count done)))
        (shift-level-pmd!! level 's  pmd-temp pmd-copy
                           :border-pmd pmd-temp)
        (*set-level-pmd-pvar level pmd-temp
                    (*!! (pmd2pvar!! pmd level)
                         (!! (aref mask x (+ y 1)))))
        (*set-level-pmd-pvar level pmd-temp
                    (+!! (pmd2pvar!! pmd-temp level)
                         (pmd2pvar!! pmd-copy level)))

        (setq count (+ count 1)))

    (if (/= ystart 0)     (setq ystart (+ ystart 1)))


    (do ((y yend)                  ;for rightward, shift east
         (x xstart  (+ x 1)))
        ((or (= x xend) (= count done)))
```

```
        (shift-level-pmd!! level 'e pmd-temp pmd-copy
                           :border-pmd pmd-temp)
        (*set-level-pmd-pvar level pmd-temp
                           (*!! (pmd2pvar!! pmd level)
                            (!! (aref mask (+ x 1) y))))

        (*set-level-pmd-pvar level pmd-temp
                           (+!! (pmd2pvar!! pmd-temp level)
                                (pmd2pvar!! pmd-copy level)))

        (setq count (+ count 1)))

      (do ((x xend)                ;for upward, shift north
           (y  yend  (- y 1)))
          ((or (= y ystart) (= count done)))
          (shift-level-pmd!! level 'n pmd-temp pmd-copy
                           :border-pmd pmd-temp)
          (*set-level-pmd-pvar level pmd-temp
                           (*!! (pmd2pvar!! pmd level)
                              (!! (aref mask x (- y 1)))))

          (*set-level-pmd-pvar level pmd-temp
                           (+!! (pmd2pvar!! pmd-temp level)
                              (pmd2pvar!! pmd-copy level)))

          (setq count (+ count 1)))

      (do ((y ystart )             ;for leftward, shift west
           (x xend  (- x 1)))
          ((or (= x (+ xstart 1)) (= count done)))
          (shift-level-pmd!! level 'w pmd-temp pmd-copy
                           :border-pmd pmd-temp)
          (*set-level-pmd-pvar level pmd-temp
                           (*!! (pmd2pvar!! pmd level)
                                (!! (aref mask (- x 1) y))))

          (*set-level-pmd-pvar level pmd-temp
                           (+!! (pmd2pvar!! pmd-temp level)
                                (pmd2pvar!! pmd-copy level)))

          (setq count (+ count 1)))

      (setq xstart (+ xstart 1))       ;set up the condition for
      (setq xend (- xend 1))           ;next circle.
      (setq yend (- yend 1))
      ) ;end of do loop
    ); end of let
  )
  ;-------------------------------------------------------------
```

```
; this function will do the convolution for all the levels of a
; pyramid using a given mask.
;
; input :  pmd-   a pyramid
;          mask-  the mask for convolution
;          m, n-  the size of the mask
;
; return:  all the levels of the input pmd get the result of
;          the convolution with the given mask
;-----------------------------------------------------------------

(*defun pmd-conv!! (pmd mask m n &optional dest-pmd)
  (*all
    (*let ((temp-pmd (make-pmd!!)))
     (do ((x (- *pmd-number-of-levels* 1) (- x 1)))
         ((< x 0))
         (*set-level-pmd x
                         temp-pmd
                         (pmd-conv-level!! pmd x mask m n))

         )
     (cond ((not dest-pmd)
            temp-pmd)
           (t
             (*set-pmd dest-pmd temp-pmd)
             dest-pmd))
)))
```

```
;-------------------------------------------------------------
; File name: make-mask.lisp
; make some arrays.
;
;-------------------------------------------------------------

(setq gaussian (make-array  '(3 3 )     :initial-contents
                            '((1 2 1) (2 4 2) (1 2 1))))

(setq laplacian1 (make-array  '(3 3 )   :initial-contents
                            '((0 1 0) (1 -4  1) (0 1 0))))

(setq laplacian2 (make-array  '(3 3 )   :initial-contents
                  '((1/4 1/2 1/4) (1/2 -3 1/2) (1/4 1/2 1/4))))

(setq arr (make-array  '(3 3 )  :initial-contents
                            '((1 1 1) (1 1 1) (1 1 1))))
```

```
;-------------------------------------------------------
; file : pmd-pref.lisp
;-------------------------------------------------------
; this function returns a value from a pyramid- pmd
; at a specific level- level at location x and y.
;-------------------------------------------------------
(defun pref-grid-level-pmd (pmd level x y)
  (*all
   (let ((level-gridsize (expt 2
                (- *pmd-number-of-levels* (1+ level)))))
     (*let ((plevel-cube (level-cube!! level))
            (p-value (pmd2pvar!! pmd level)))
           (cond ((and (>= x 0) (< x level-gridsize)
                       (>= y 0) (< y level-gridsize))
                  (pref p-value (pref-grid plevel-cube x y)))
         (t (format t " x or y out of range")))
))))

;-------------------------------------------------------------
; this function returns a pyramid, in which, at level- level,
; each pixel contains the value from original pyramid- pmd at
; relative location  (rx,ry)
;-------------------------------------------------------------
(*defun pref-grid-level-pmd-relative!! (pmd level rx ry)
  (*all
    (*let ((temp-pmd (make-pmd!!)))
      (let ((tx 0) (dx 0)
            (ty 0) (dy 0))

        (cond ((> rx 0) (setq tx  (- rx 1)) (setq dx 1))
              (t (setq tx (- 1 rx)) (setq dx 0)))
        (cond ((> ry 0) (setq ty  ry) (setq dy 1))
              (t (setq ty (- 0 ry)) (setq dy 0)))

        (cond ((= dx 1)
               (shift-level-pmd!! level 'w pmd temp-pmd))
              (t (shift-level-pmd!! level 'e pmd temp-pmd)))

        (dotimes (x tx)
          (cond ((= dx 1)
        (shift-level-pmd!! level 'w temp-pmd temp-pmd))
       (t (shift-level-pmd!! level 'e temp-pmd temp-pmd))))

        (dotimes (y ty)
          (cond ((= dy 1)
        (shift-level-pmd!! level 'n temp-pmd temp-pmd))
       (t (shift-level-pmd!! level 's temp-pmd temp-pmd))))

        temp-pmd))))
```

```
;-------------------------------------------------------------
;file:  pmd-edges.lisp
;-------------------------------------------------------------

;-------------------------------------------------------------
; input:  start-level--> a level to start to find edges in pmd
;          thresh    --> a thresh value for the pmd
;          pmd       --> a pyramid to find its edges
;          dest-pmd  --> a pyramid to record the edges of pmd.
; returns: a pyramid recording the edges.
;-------------------------------------------------------------
(*defun edges-pmd!! (start-level thresh pmd &optional dest-pmd)
   (*let ( (parent-pmd (make-pmd!!))
           (edge-pmd (make-pmd!!)))

;       start out by setting parent-value to threshold so all
;       edges  will be calculated for the start level

        (*set-level-pmd-pvar start-level parent-pmd (!! thresh))

        (refine pmd start-level parent-pmd edge-pmd thresh)

        (cond ((not dest-pmd)
               edge-pmd)
              (t
                (*all (*set-pmd-let dest-pmd edge-pmd))))

));end *find-edges

;-------------------------------------------------------------
; This function will recursively go down the pyramid from
; start-level to the 0th level. At each level, if the parent's
; value was above the threshold, it calculates the edge value
; for each pixel (on that level) and stores this value in the
; temporary pmd 'edge.
;-------------------------------------------------------------
(*defun refine (pmd level parent edge thresh)
   (*all  (*when (and!! (=!! (!! level)  *pmd-level-number*)
                  (>=!! (pmd2pvar!! parent level) (!! thresh)))
   (*set-level-pmd-pvar level edge (edge-op!!  pmd  level))))

; As long as you haven't reached the 0th level, keep refining
; Refine by setting the parent-value of children to edge value
; of current level
   (cond (  (/= level 0)
            (*all
              (send-level-children-pmd!! level edge parent)
              (refine pmd (- level 1) parent edge thresh)) )
         (t 't))
```

```
);end refine

;-------------------------------------------------------
;this will calculate  (A - B)^2  +  (X - Y)^2  where:
;
;         +-----------------+
;         |   Y    |    B   |
;         +--------+--------+
;         |   A    |    X   |
;         +--------+--------+
;-------------------------------------------------------
(*defun edge-op!! (pmd level)
 (*all
  (*let (  (temp-pvar1 (!! 0))
           (temp-pvar2 (!! 0)) )

        (*all (*when (=!! *pmd-level-number* (!! level))
                     (*shift 's level (pmd2pvar!! pmd level)
                              temp-pvar1)))

        (*all (*set temp-pvar2 temp-pvar1))

; First set temp-pvar1 to start value - value from
; north and east.(A-B)
       (*all (*when (=!! *pmd-level-number* (!! level))
                    (*shift 'w level temp-pvar1
                            temp-pvar1)))
       (*all (*set temp-pvar1
         (-!! (pmd2pvar!! pmd level) temp-pvar1)))

; Second set temp-pvar2 to start value - value from
; north and west.(X-Y)
        (*all (*when (=!! *pmd-level-number* (!! level))
                     (*shift 'e level temp-pvar2
                             temp-pvar2)))
        (*all (*set temp-pvar2
                (-!! (pmd2pvar!! pmd level) temp-pvar2)))

    ;send back this value...(A-B-let)^2 + (X-Y)^2
        (*all (*when (=!! *pmd-level-number* (!! level))
                  (*shift 'w level  (*!! temp-pvar2 temp-pvar2)
                             temp-pvar2)))
        (*all (*set
                temp-pvar1
                (+!! (*!! temp-pvar1 temp-pvar1)
                     temp-pvar2)))
        (*all (pref!! temp-pvar1 *pmd-self-address*))
)));end edge-op!!
```

```
;-------------------------------------------------------------
; file: pmd-load.lisp
;
; routines for loading images (UT formatted files)
; into pvars (via 2-d array - need machine configuration
; to be at least as large as this array)
; - see also unload.lisp
;
; returns pvar which contains image data
;-------------------------------------------------------------
(defun read-file-pmd!! (file-top file-leaf &optional dest-pmd)
 (*all
  (*let  (( temp-pmd (make-pmd!!)))
         (cond ((not dest-pmd)
                (setf (pmd-top!!  temp-pmd)
                      (read-file-pvar!! file-name))
                (setf (pmd-leaf!!  temp-pmd)
                      (read-file-pvar!! file-name))
                temp-pmd)
               (t
                 (setf (pmd-top!!  dest-pmd)
                       (read-file-pvar!! file-name))
                 (setf (pmd-leaf!! dest-pmd)
                       (read-file-pvar!! file-name))
                 dest-pmd))
)))


(defun read-file-pvar!! (file-name &optional dest-pvar)
; Open file for reading bytes, if file
; doesn't exit returns nil
  (setq port (open file-name
                   :direction  :input
                   :element-type '(unsigned-byte 8)
                   :if-does-not-exist nil))
  (cond (   (eq nil port) (print "file not opened-does not
                                  exist")(terpri))
        (t  (print "file opened") (terpri)
    ; lop-off-header will send back the dimensions of the image
; this info is given in the header of the file
            (lop-off-header port);sets dimension-row &
            ;     dimension-col
            (cond ((not dest-pvar)
             (read-file2 port dimension-row dimension-col))
                (t (*set dest-pvar
                 (read-file2 port dimension-row dimension-col))
                     dest-pvar))
)))
```

```
; read image data from file and put into global array image
(defun read-file2 (port dim-row dim-col)
   (cond ((or (> dim-row (dimension-size 0))
              (> dim-col (dimension-size 1)))
          (print "Image too large for given machine size"))
         (t
          (let ((image (make-array (cons (dimension-size 0)
                                         (list (dimension-size 1)))
                                   :initial-element 0)))
          (do  (  (y 0 (+ y 1)) )
               (  (= y dim-row) 't)
               (do (  (x 0 (+ x 1)) )
                   (  (= x dim-col) 't)
                      (setf (aref image x y)
                      (read-byte port nil 'eof) )
                );end do x
           );end do y
           (close port)
           (array-to-pvar-grid image))));end let,t & cond
);end read-file2

(defun lop-off-header (port)
; The dimensions of the image are:1st byte
; + 256*2nd byte=# of rows :3rd byte + 256*4th byte=#of  cols
       (setq dimension-row (+ (read-byte port)
                              (* 256 (read-byte port))));#of rows
       (setq dimension-col (+ (read-byte port)
                              (* 256 (read-byte port))));#of cols
;then there are 507 bytes worth of junk
       (do ( (x 0 (+ 1 x))) ;inits
           ( (= x 507) 't)  ;until end.send back 't
           (read-byte port));throw away
;last (or 512th byte) is type of image it is
;(setq type-of-image 'c)
     (setq type-of-image (read-byte port))
     (format t
      "the type of picture is  ~c" (int-char type-of-image))
);end lop-off-header


(*defun pg(x) (ppp x :mode :grid))
(defun pa(a n m)
  (do ((j 0 (+ j 1))) ((= j n))
     (do ((k 0 (+ k 1))) ((= k m))
          (format t "~3d  " (aref a j k)))
     (format t "~%")))

(*defun *load-level (source-pvar dest-pvar ilevel)
     (*all (*load-level2 source-pvar dest-pvar ilevel)))
```

```
(*defun *load-level2 (source-pvar dest-pvar ilevel)
  (let ( (level-gridsize (expt 2 (- num-levels ilevel))))
     (*let ((plevel-cube (level-cube!! ilevel)))
           (dotimes (x level-gridsize)
              (dotimes (y level-gridsize)
              (setf (pref dest-pvar (pref-grid plevel-cube x y))
                      (pref-grid source-pvar x y))))

)));end load-level2
```

```
;-------------------------------------------------------------
; file: unload.lisp
;
; routines for unloading a pvar into an image file
; (UT format)
;
; returns nil if file made properly
;-------------------------------------------------------------
(defun write-file-pmd (pmd file-top file-leaf row-size col-size)
 (*all
   (write-file-pvar (pmd-top!! pmd)
        file-top row-size col-size)
   (write-file-pvar (pmd-leaf!! pmd)
        file-leaf row-size col-size)
))


(defun write-file-pvar (p-img file-name row-size col-size)
; open file for writing bytes, if file already exists then new
; file will have larger version number (p420)
; closes file automatically at function end
       (setq port (open file-name
                        :direction  :output
                        :element-type '(unsigned-byte 8)))
       (cond (   (eq nil port) (print "file not opened-does not
                                       exist")(terpri))
             (t  (print "file opened") (terpri)
; write header to file
                 (write-header port row-size col-size)
; write image data to file
                 (write-file2 port p-img row-size col-size)
))) ; end of write-file


; write image data from file and put into global array image
(defun write-file2 (port p-img row-size col-size)
  (let ((image (make-array (cons row-size (list col-size)))))
       (pvar-to-array-grid p-img image)
       (do  (  (y 0 (+ y 1)) )
            (  (= y row-size) 't)
            (do (  (x 0 (+ x 1)) )
                (  (= x col-size) 't)
                (setf (aref image x y)
                      (write-byte (aref image x y) port) )
             );end do x
        );end do y
       (close port)
));end write-file2
```

```
(defun write-header (port row-size col-size)
;the dimensions of the image are:1st byte
;                      + 256*2nd byte=# of rows
;                    :3rd byte + 256*4th byte=#of  cols
       (write-byte (mod row-size 256) port)
                   ; lsb of rowsize
       (write-byte (floor (/ row-size 256)) port)
                   ; msb of rowsize
       (write-byte (mod col-size 256) port)
                   ; lsb of colsize
       (write-byte (floor (/ col-size 256)) port)
                   ; msb of colsize
;then there are 507 bytes worth of junk
       (do ( (x 0 (+ 1 x))) ;inits
           ( (= x 507) 't)  ;until end.send back 't
           (write-byte 0 port));throw away
;last (or 512th byte) is type of image it is (byte image)
       (write-byte (char-int #\b) port)
);end write-header
```

```
;-------------------------------------------------------
; file: pmd-init.lisp
;-------------------------------------------------------
;-------------------------------------------------------
; This is my file for initializing the environment for
; testing the pyramid emulation
;-------------------------------------------------------
(defun test()
(load "pyramid-emulate.lisp")
(pyramid-emulate )
)


(defun init()
   (*defpmd p1)
   (*defpmd p2)
   (*defpmd p3)

   (*defpmd p4 )
   (*defpmd p5 )
   (*defvar self (self-address!!))
   (*defvar a1)
   (*all (*set a1 (if!! (>!! self (!! 8200))
                        (!! 255)
                        (!! 0))))
   (*set-pmd-pvar p1 a1)
   (*set-pmd-pvar p4 self)
   (*set-pmd-pvar p5 (assign-levels!!))
)

;;; Lucid Common Lisp, Development Environment Version 2.1,
;;; 6 December 1987
;;; Copyright (C) 1987 by Lucid, Inc.  All Rights Reserved
;;;
;;; This software product contains confidential and trade secret
;;; information belonging to Lucid, Inc.  It may not be copied
;;; for any reason other than for archival and backup purposes
;;; Connection Machine Software, Release 4.3
;;;
;;; Copyright (C) 1988 by Thinking Machines Corporation.
;;; All rights reserved.
> (in-package '*lisp)
#<Package "*LISP" A07023>
> (cm:attach :16k)
;;; Loading source file "/usr/local/etc/cm_configuration.lisp"

 Warning:  If you are using *Lisp, you must now do *COLD-BOOT.
16384
> (*cold-boot :initial-dimension '(128 128))
```

```
Loading microcode functions file
"/usr/local/lib/cm/microcode-beta-f4305/PARIS-1-1-UCODE-FORMS"
 . . . Done.
Loading microcode ucode control store file
"/usr/local/lib/cm/microcode-beta-f4305/PARIS-1-1-UCODE-CS.CS"
 . . . Done.
;;; Expanding Dynamic Memory
T
> (load "pyramid-emulate.lisp")
;;; Loading source file "pyramid-emulate.lisp"

;;; Expanding Reserved Memory
;;; GC: 417452 words [1669808 bytes] of dynamic storage in use.
;;; 580946 words [2323784 bytes] of free storage available
;;; before a GC.
;;; 1579344 words [6317376 bytes] of free storage available if
;;; GC is disabled. Expanding Reserved Memory
;;; GC: 417452 words [1669808 bytes] of dynamic storage in use.
;;; 580946 words [2323784 bytes] of free storage available
;;; before a GC.
;;; 1579344 words [6317376 bytes] of free storage available
;;; if GC is disabled.
;;; GC: 417452 words [1669808 bytes] of dynamic storage in use.
;;; 580946 words [2323784 bytes] of free storage available before
;;; a GC.
;;; 1579344 words [6317376 bytes] of free storage available if GC
;;; is disabled.
;;; While compiling PSET-PMD-TOP
;;; Warning: No source code is available for inline expansion of
;;; call to *LISP-I::NEQ

#P"/u1/columbia/lbrown/ju/pyramid/pyramid-emulate.lisp"
> (pyramid-emulate)


 **** Pyramid Emulation *****

      -- version 1.0 --

T
> (*defpmd p1 (pmd!! 1))
P1
> (*defpmd p2 (pmd!! 0))
P2
> (*defpmd p3)
P3
> (*set-pmd-pvar p3 (self-address!!))
NIL
> (*defpmd p4)
```

```
P4
> (*set-pmd-pvar p4 (assign-levels!!))
NIL
> (*display-pmd-part p3 4 7)

 Level  4 :

   248    504   1272   1528   4344   4600   5368   5624
   760   1016   1784   2040   4856   5112   5880   6136
  2296   2552   3320   3576   6392   6648   7416   7672
  2808   3064   3832   4088   6904   7160   7928   8184
  8440   8696   9464   9720  12536  12792  13560  13816
  8952   9208   9976  10232  13048  13304  14072  14328
 10488  10744  11512  11768  14584  14840  15608  15864
 11000  11256  12024  12280  15096  15352  16120  16376

 Level  5 :

  1008   2032   5104   6128
  3056   4080   7152   8176
  9200  10224  13296  14320
 11248  12272  15344  16368

 Level  6 :

  4064   8160
 12256  16352

 Level  7 :

 16320
NIL
> (*display-pmd-part p4 4 7)

 Level  4 :

     4      4      4      4      4      4      4      4
     4      4      4      4      4      4      4      4
     4      4      4      4      4      4      4      4
     4      4      4      4      4      4      4      4
     4      4      4      4      4      4      4      4
     4      4      4      4      4      4      4      4
     4      4      4      4      4      4      4      4
     4      4      4      4      4      4      4      4

 Level  5 :

     5      5      5      5
     5      5      5      5
```

```
     5       5       5       5
     5       5       5       5

 Level  6 :

     6       6
     6       6

 Level  7 :

     7
NIL
> (shift-level-pmd!! 4 'n p4 p1 :border-pmd p3)
#<GENERAL-Pvar P1 63739-222>
> (*display-level-pmd p1 4)

     4       4       4       4       4       4       4       4
     4       4       4       4       4       4       4       4
     4       4       4       4       4       4       4       4
     4       4       4       4       4       4       4       4
     4       4       4       4       4       4       4       4
     4       4       4       4       4       4       4       4
     4       4       4       4       4       4       4       4
  8440    8696    9464    9720   12536   12792   13560   13816
NIL
> (shift-level-pmd!! 4 'n p3 p1)
#<GENERAL-Pvar P1 63739-222>
> (*display-level-pmd p1 4)

   760    1016    1784    2040    4856    5112    5880    6136
  2296    2552    3320    3576    6392    6648    7416    7672
  2808    3064    3832    4088    6904    7160    7928    8184
  8440    8696    9464    9720   12536   12792   13560   13816
  8952    9208    9976   10232   13048   13304   14072   14328
 10488   10744   11512   11768   14584   14840   15608   15864
 11000   11256   12024   12280   15096   15352   16120   16376
     0       0       0       0       0       0       0       0
NIL
> (pmd-conv-level!! p3 4 arr 3 3 p1)
#<GENERAL-Pvar P1 63739-222>
> (*display-level-pmd p1 4)
;;; GC: 556792 words [2227168 bytes] of dynamic storage in use.
;;; 441606 words [1766424 bytes] of free storage available before
;;; a GC.
;;; 1440004 words [5760016 bytes] of free storage available if
;;; GC is disabled.

 57016   56248   60088   71608   81592   93112   96952   79800
 14520   13752   17592   29112   39096   50616   54456   37304
```

```
 22200  21432  25272  36792  46776  58296  62136  44984
 45240  44472  48312  59832  69816  81336  85176  68024
 65208  64440  68280  79800  89784 101304 105144  87992
 88248  87480  91320 102840 112824 124344 128184 111032
 95928  95160  99000 110520 120504 132024 135864 118712
 78008  77240  81080  92600 102584 114104 117944 100792
NIL
> (sys:quit)
64.5u 10.4s 20:05 6% 7344+19414k 926+9io 797pf+0w
cm3:exitConnection closed by foreign host.
ju>
script done on Wed May  4 17:36:59 1988
```

References

1. Bajscy, R., and Lieberman., L. "Gradient as a Depth Cue". *Computer Graphics and Image Processing 16* (1976), 52-67.

2. Blake, A., and Zisserman, A.. *Visual Reconstruction.* Cambridge, Mass, MIT Press, 1987.

3. Brown, L.G., Ju, Q., and Norman, C. User's Manual for Pyramid Emulation on The Connection Machine. Columbia University, April, 1988.

4. Choi, D. J. *Solving the Depth Interpolation Problem on a Parallel Architecture with the Efficient Numerical Methods.* Ph.D. Th., Department of Computer Science, Columbia University, 1988.

5. Choi, D.J., and Kender, J.R. Solving the Depth Interpolation Problem on a Parallel Architecture with a Multigrid Approach. Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, IEEE, 1988, pp. .

6. Davis, L.S., Janos, L., and Dunn, S.M. "Efficient Recovery of Shape from Texture". *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-5*, 5 (September 1983).

7. Gibson, J.J.. *The Perception of the Visual World.* Houghton-Mifflin, Boston, Mass., 1950.

8. Grimson, W.E.L. "An implementation of a computational theory of visual surface interpolation". *Computer Vision, Graphics, Image Processing 22* (1983).

9. Ibrahim, H., Kender, J., Band rown, L. Parallel Vision Algorithms - Annual Technical Report. Columbia University, November, 1987.

10. Ibrahim, H. A. H. On the Implementation of Pyramid Algorithms on the Connection Machine. Proceedings of DARPA Image Understanding Workshop, April, 1988, pp. 634-640.

11. Kender, J.R. *Shape from Texture.* Ph.D. Th., C.M.U., 1980.

12. Kjell, B.P., and Dyer, C.R. "Segmentation of Textured Images". *Computer Vision and Pattern Recognition 35* (1986).

13. Laws, K.I. *Textured Image Segmentation.* Ph.D. Th., University of Southern California, Janurary 1980.

14. Moerdler, M.L., and Kender, J.R. An Integrated System That Unifies Multiple Shape From Texture Algorithms. Proceedings of American Association of Artificial Intelligence 87, 1987.

15. Moerdler, M.L. *Shape-From-Textures:A Paradigm for Fusing Middle Level Vision Cues.* Ph.D. Th., Columbia University, Department of Computer Science, in process due 1988.

16. Ohta, Y., Maenobu, K., and Sakai, T. Obtaining Suface Orientation from Texels under Perspective Projection. Proceedings of the Seventh International Joint Conference on Artificial Intelligence, IJCAI, 1981.

17. Raafat, H.M., and Wong, A.K.C. "Texture-Based Image Segmentation". *Computer Vision Pattern Recognition 35* (1986).

18. Terzopoulos, D. *Multiresolution Computation of Visible-Surface Representations.* Ph.D. Th., Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1984.

19. Witkin, A.P. Recovering Surface Shape from Orientation and Texture. In *Computer Vision*, North-Holland Publishing Company, 1980, pp. 17-45.

20. Witkin, A.P. "Recovering Surface Shape and Orientation from Texture". *Artificial Intelligence* (August 1981), 17-45.